
Technical report, IDE0602, January 2006

Wireless Sensor Network Simulator

Master's Thesis in Electrical Engineering

Thammakit Sriporamanont and Gu Liming



School of Information Science, Computer and Electrical Engineering
Halmstad University

Wireless Sensor Network Simulator

Master's thesis in Electrical Engineering

School of Information Science, Computer and Electrical Engineering
Halmstad University
Box 823, S-301 18 Halmstad, Sweden

January 2006

Preface

This Master's thesis is the final step of the Master of Science Degree in Electrical Engineering at Halmstad University, Sweden.

We would like to express our gratitude to our supervisor Prof. Tony Larsson for his support and helpful suggestions through out the work on this thesis. We would also like to thank Latef Berzenji for his valuable help on the language of this thesis.

Finally, many thanks go to our parents and our friends for their encouragement.

Thammakit Sriporamanont and Gu Liming
Halmstad University, January 2006

Abstract

In the recent past, wireless sensor networks have been introduced to use in many applications. To design the networks, the factors needed to be considered are the coverage area, mobility, power consumption, communication capabilities etc. The challenging goal of our project is to create a simulator to support the wireless sensor network simulation. The network simulator (NS-2) which supports both wire and wireless networks is implemented to be used with the wireless sensor network. This implementation adds the sensor network classes which are environment, sensor agent and sensor application classes and modifies the existing classes of wireless network in NS-2. This NS-2 based simulator is used to test routing protocols – Destination-Sequenced Distance Vector (DSDV), and Ad-Hoc On-Demand Distance Vector (AODV) as one part of simulations. Finally, the sensor network application models and the extension methods of this NS-2 based simulator for simulations in specific wireless sensor network applications are proposed.

Contents

1	INTRODUCTION	1
1.1	GOALS.....	1
1.2	METHOD	2
2	BACKGROUND AND RELATED WORK	3
2.1	A SURVEY ON WIRELESS SENSOR NETWORKS	3
2.2	NETWORK SIMULATOR (NS).....	5
2.3	RELATED WORKS	7
3	IMPLEMENTATION OF NS-2.28 BASED WIRELESS SENSOR NETWORK SIMULATOR.....	9
3.1	BASIC STRUCTURE OF WIRELESS SENSOR NETWORKS.....	9
3.2	SIMULATION MODEL OF WIRELESS SENSOR NETWORKS.....	10
3.3	IMPLEMENTATION SENSOR NETWORK IN NS-2.28	12
3.4	EXAMPLE OF USING THE SIMULATOR TO TEST ROUTING PROTOCOLS	22
4	EXTENSION OF SIMULATOR FOR SPECIFIC SENSOR NETWORK APPLICATIONS.....	27
4.1	SPECIFIC SENSOR NETWORK APPLICATION MODELS.....	27
4.2	EXTENSION METHOD OF THE NS-2 BASED SIMULATOR FOR SPECIFIC APPLICATION.....	29
5	CONCLUSION	35
5.1	DISCUSSION.....	35
5.2	FUTURE WORK.....	36
6	REFERENCES	37
	APPENDIX A BASIC WIRELESS NETWORK MODEL IN NS-2	39
A1	MOBILENODE	39
A2	NODE MOVEMENTS	41
A3	NETWORK COMPONENTS IN MOBILENODES	41
A4	DIFFERENT TYPES OF ROUTING AGENTS IN MOBILE NETWORKING	43
A5	TRACE SUPPORT	43
A6	WIRELESS TRACE FORMATS	43
	APPENDIX B IMPLEMENTATION CODES FOR WIRELESS SENSOR NETWORK CLASSES	47
B1	ENVIRONMENT MODULE COMMAND().....	47
B2	ENVIRONMENT MODULE EMANATE().....	49
B3	NS-MOBILENODE.TCL	50
B4	CMU-TRACE.H,CC	52
	APPENDIX C EXAMPLE TCL CODES FOR TESTING ROUTING PROTOCOLS.....	55

1 Introduction

In recent years, advance in micro-electro-mechanical systems (MEMS) which are the integrations of mechanical elements, sensors, actuators and electronics on a common silicon substrate through micro fabrication technology and have enabled low-cost and low-power in electronics and wireless communication technology enabled tiny sensor nodes to communicate in short distance. Wireless Sensor Networks (WSNs) comprise numerous tiny sensor nodes that are deployed in spatially distributed terrain. Each sensor node is endowed with a limited amount of processing, but when coordinated with the information from other nodes, they have the ability to measure the given physical environment in great details or to execute a task with complex functions. Hence, a sensor network can be described as a collection of sensor nodes that coordinate with each other to perform some specific actions. Since each sensor node is fitted with an on-board processor, sensor nodes use their processing abilities to find out simple computations and transmit only the required data. These features allow the sensor networks to use in many applications, like military, security and environment. Wireless sensor networks can also be deployed in the ways that the wired sensor system cannot be deployed such as in the chemical environments that are inaccessible by humans.

Optimizing a wireless sensor network also bring many open issues in the network design. Normally, analyzing methods, computer simulation, and physical measurement are the three main techniques used for analyzing the performance of wired and wireless networks. However, in wireless sensor networks, they contain a lot of constrains, such as energy limitation, decentralized collaboration, and fault tolerance. These constraints cause many unresolved design and implementation problems, so measurements are virtually impossible. It appears that simulation is currently the primary feasible approach for analysing the behaviours of sensor network and reducing the development lifetime.

In our project, NS-2 [1] is chosen as our simulation environment because it is the most widely used network simulator. NS-2 has several key benefits which suit for doing the simulation:

- It provides extensive support for simulating TCP/IP, Routing and multicast protocols over wired and wireless network.
- It provides a lot of standard modules to be used in sensor network.
- It is an object oriented design which provides a lot of documents.
- It uses Tcl to specify the components and OTcl to glue them together.

1.1 Goals

As wireless sensor networks have been used in a lot of applications, monitoring nodes in the network are needed for some applications. The important functions which need to be monitored are the communicated data between each node, the movement of nodes, etc. The goal of this project is to design a simulator which can be used to monitor wireless sensor and actuator networks on a useful level to evaluate different functions, cooperation patterns, network topologies, and physical space, time and event scenarios. Though, there are a lot of routing protocols which can be used in the wireless sensor networks, this simulator is used to test

different routing algorithms as one part of simulation. Furthermore, this simulator is extended for the simulation of sensor network applications such as enemy surveillance application and chemical gas cloud application. The results of the simulation can be shown in both text and graphical interface for the events and for the illustration of network topology with the physical environment as background.

1.2 Method

To achieve the goals of the project, the work is divided into four parts: studying NS-2, definition the structure of wireless sensor networks, implementation of sensor network in the NS-2 environment, and extension to simulate in specific applications. In the first part, we study how NS-2 is used in programming, and create some simple wireless network simulations, while in the second part, we define the structure of wireless sensor networks, and in the third part, we implement NS-2 by adding some components to use in the sensor network. Finally, our simulator is extended to simulate the specific applications.

2 Background and Related Work

In this chapter, a survey of wireless sensor networks, the network simulator (NS) and the related work are discussed.

2.1 A survey on wireless sensor networks

The recent developments in micro-electro-mechanical system (MEMS) and wireless communication technology make wireless sensor networks more interested. Tiny sensor nodes, which are low-cost and low-power, comprise sensing, data processing and communicating components. A large number of sensor nodes generate the sensor networks. Sensor nodes have a self-organizing capability to communicate to their neighbor nodes. Before they transmit the data to their neighbor, they use their data processing component to find out the simple computations and select the required data to transmit. With these features, the wireless sensor network can be used in many applications. In this part, some sensor network applications are discussed. [2]

2.1.1 Military applications

The initial wireless sensor network is to be used in the military applications. Since sensor nodes are low-cost, destruction of some nodes by hostile actions in the battlefields may not affect a military operation. The features of robustness, self-organizing and fault tolerance make sensor networks appropriate for military use. Distributed sensing has the advantages of being able to provide redundant and hence highly reliable information on threats as well as the ability to localize threats by both coherent and incoherent processing among the distributed sensor nodes. Examples of the military applications of sensor networks are monitoring army, equipment and ammunition [2] and enemy surveillance.

Monitoring army, equipment and ammunition

Every troop, equipment, vehicle and critical ammunition is attached with sensors. In the battlefield, the commanders can monitor the status of their armies, equipment, and ammunitions from reported data which are generated constantly by sensors and forwarded to the commanders.

Enemy surveillance

The sensor network deploys heterogeneous collections of sensors capable of observing and reporting on various dynamic properties of critical terrains in a timely manner. Data reports from areas of the sensor network will be aperiodic and diverse, carrying a range of application specific data.

2.1.2 Environment applications

Examples of environment applications of sensor networks are forest fire detection [2] and flood detection [3].

Forest fire detection

The sensor network densely deploys a lot of sensor nodes distributed in a wide forest area. These nodes are integrated with radio frequency system and may be supplied with power from solar cells which can be used for a long time. When the nodes detect the fire, they report the data to the central station. This application needs the real-time communication before the fire spreads and becomes uncontrollable.

Flood detection

An example of the flood detection is the ALERT [4], which was developed by the National Weather Service in the 1970s. ALERT provides important real-time rainfall and water level information to evaluate the possibility of potential flooding. ALERT sensors are equipped with rainfall, water level and weather sensors. The detected data are reported via light-of-sight radio communication from sensor site to the central station.

2.1.3 Indoor applications

Indoor applications could be home or office applications. In this part, the home automation [2] and open secure office [5] applications are discussed.

Home automation

The tiny sensor nodes could be attached to the electrical appliances such as air-conditioner, computer, refrigerator etc. These sensor nodes can communicate with each other or with the users who live outside this network. The users can control these appliances via the internet or satellite network.

Open secure office

The sensor nodes are attached to the valuable equipments such as laptops, printers, PDAs etc. Each room has a local base station and, at the entrance door, the main base station is installed. When someone tries to move the equipment, the sensor at that equipment, which could be a vibration sensor, sends an alarm to its local base station which will send a command to the camera to take a picture of that room. Or when the equipment is moved near the entrance door, the main base station detects an alarm and sends a command to lock the door automatically.

2.2 Network Simulator (NS)

The network simulator (NS), which is a discrete event simulator for networks, is a simulated program developed by VINT (Virtual InterNetwork Testbed) project group (A Collaboration among USC/ISI, Xerox PARC, LBNL, and UCB). It supports simulations of TCP and UDP, some of MAC layer protocols, various routing and multicast protocols over both wired and wireless network etc. The basic structure of NS-2 is:

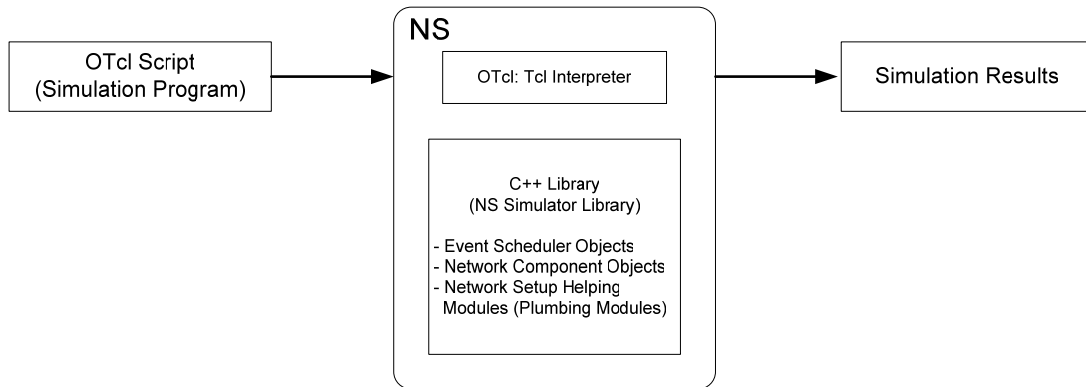


Figure 2.1: The basic structure of NS

To setup and run a simulation, a user writes an OTcl script, which is a simulation program to initiate an event scheduler, set up the network topology using the network objects and plumbing functions in the library, and to tell traffic sources when to start and stop transmitting packets through the event scheduler. When NS-2 which works as OTcl interpreter receives the OTcl script, it will set environment parameters following the received script. If a user wants to make a new network object, it will be easy to make a compound object from the object library, and plumb the data path through the object rather than write a new one. When the simulation is finished, the simulation results are produced in one or more text-based output files that contain detailed simulation data, which can be used to analyze directly or can be used in the graphical user interface “Network Animator (NAM)” [6]. This graphical user interface shows the simulation result in an easy way.

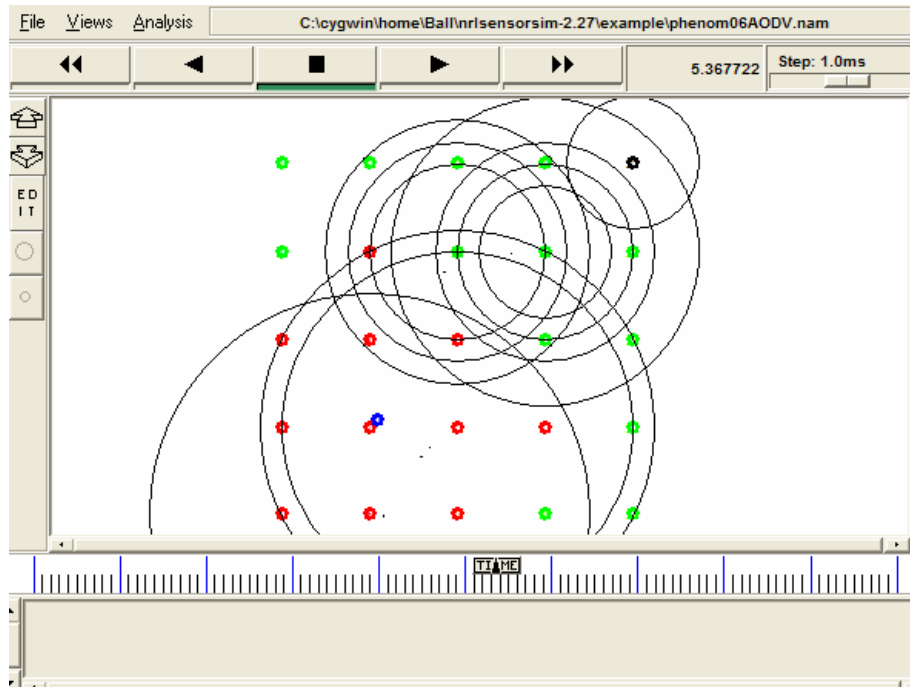


Figure 2.2: Network Animator (NAM)

The language that is written in NS-2 is not only OTcl but also C⁺⁺. The event scheduler and the basic network component objects in the data path are written and compiled using C++ to reduce packet and event processing time. These compiled objects need the OTcl linkage to create a matching OTcl object for each of the C++ objects to be able to work with OTcl interpreter.

NS-2 is a free software which can be downloaded from [1]. It can run both in Unix and Windows. But to run in Windows, it requires installing Cygwin [7] to set the computer environment as UNIX environment. Details can be found in [8], [9], [10] and [11].

2.3 Related Work

The related works of wireless sensor networks simulation are discussed in this section.

2.3.1 Wireless Sensor network simulator

There are several simulators that have been used in sensor network research. In this part, we discuss some simulators that can be used to evaluate the wireless sensor networks.

SensorSim

SensorSim [12], which has been built on NS-2, is a simulation framework for sensor networks. It provides sensor channel models, energy consumers, lightweight protocol stacks for wireless micro sensors, scenario generation and hybrid simulation. The sensor channel models the dynamic interaction between the sensor nodes and the physical environment. At each node, energy consumers are said to control the power in multiple modes to efficiently use the power and prolong the nodes' lifetime. But this simulator is no longer developed, therefore, no more available.

J-Sim

J-Sim (formerly known as JavaSim) [13], is an object-oriented, component-based, compositional simulation environment written in Java. The key benefits of J-Sim are that modules can be easily added and deleted in a plug-and-play manner, and it is also useful both for network simulation and emulation by incorporating one or more real sensor devices. J-Sim provides supporting target, sensor and sink nodes, sensor channels and wireless communication channels, physical media, power models and energy models.

GloMoSim

GloMoSim (Global Mobile Information system Simulate) [14], which is a scalable simulation environment written both in C and Parsec, is capable of parallel discrete-event simulation. GloMoSim is a collection of library modules each of which simulates a specific wireless communication protocol in the protocol stack. GloMoSim is widely used to simulate Ad-hoc and Mobile wireless networks.

SENS

SENS (Sensor, Environment, and Network Simulator) [15] is a high level sensor network simulator for wireless sensor network applications. It consists of application, network and physical components which are interchangeable and extensible. Application components are used to simulate the execution on a sensor node, network components are used to simulate the packet to send and receive functions of a sensor node, and physical components are used to model sensors, actuators and power and interact with the environment. Users can modify these existing

components or write the new one for their applications, network models, sensor capabilities and environments. SENS features the application portability to enable direct portability between the simulator and real sensor nodes.

TOSSIM

TOSSIM [16] is a discrete event simulator built specifically to simulate the Berkeley MICA mote hardware platform running applications built on TinyOS wireless sensor networks. Instead of running a TinyOS application on motes, users can compile it in the TOSSIM framework which runs on a PC. By running on a PC, users can test the application in a controlled environment and develop their TinyOS codes by using debuggers and other development tools. TOSSIM supports four key requirements of a TinyOS simulator which are scalability, completeness, fidelity and bridging. TOSSIM is sufficient for the evaluation of a high level application, but it is not sufficient for low level protocol such as MAC. Due to TOSSIM mainly focuses on the TinyOS sensor networks, it should not be used for absolute evaluations in the real world.

OMNeT++

OMNeT++ (Objective Modular Network Test-bed in C++) [18] is a wireless sensor network simulator which is based on the discrete event simulation framework. As the OMNeT++ model collects hierarchy modules, it is possible to capture the complex system. These modules are separated into two types which are simple and compound. Simple modules are programmed in C++, while compound modules which consist of simple modules are programmed in a high-level language (NED). To communicate between modules, the exchanging messages which represent the packets in the network are used. OMNeT++ supports users to study the effect of scale, node-level architecture, energy efficiency, communication architecture, system architecture, protocols etc.

2.3.2 Simulation of wireless sensor networks

In [12], a versatile environment was constructed and three types of nodes: sensor, target and user nodes were defined for studying sensor networks. This study focuses on power consumption and battery lifetime. In [19], the NS-2 framework was extended to support for simulating sensor networks. PHENOM routing protocol was created to broadcast PHENOM packet from PHENOM node (same as target node mentioned in [12]) to the sensor node or between the PHENOM nodes. This extended framework was used to examine routing protocols in a dynamic sensor network.

3 Implementation of NS-2.28 based wireless sensor network simulator

The network simulator (NS-2) supports the simulation of wireless networks, but, for the simulation of wireless sensor networks, it needs to be extended. The processes to extend NS-2.28 (Network Simulator version 2.28) for wireless sensor network simulations are shown in this chapter. First, the basic structure and the simulation model of wireless sensor networks are examined, and then the NS-2.28 based wireless sensor network simulator is implemented. Finally, an example of using this simulator to test routing protocols is given.

3.1 Basic Structure of Wireless Sensor Networks

Wireless networks consist of numerous mobile nodes which communicate with each other via wireless channels, while in wireless sensor networks, these mobile nodes are attached with sensors to sense the physical target and processors to generate data packets from sensing data and transmit them to their neighbors. Mobile nodes in wireless sensor networks are classified into two types, sensor and sink nodes.

Sensor node: This is a mobile node moving freely to monitors the physical environment. Once it detects its physical target, it generates a data packet and sends it to the sink node via the wireless channel. The processor in the sensor node may be set the threshold value to compare with the detected data before it generates and sends a data packet.

Sink node: This node collects all data packets from sensor nodes. Users use these collected data to analyze their targets.

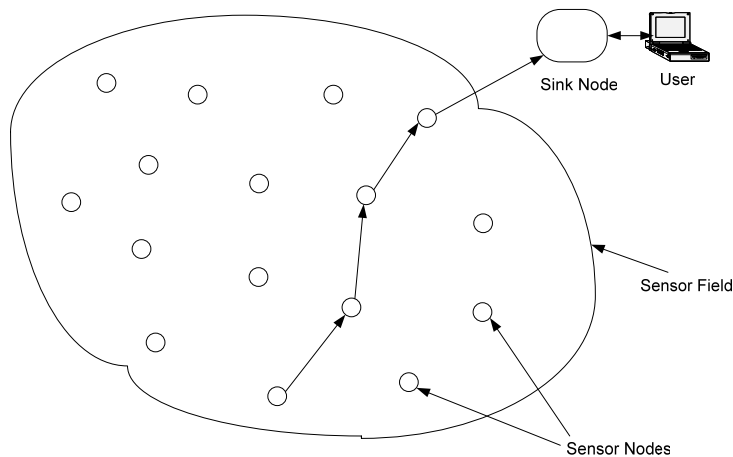


Figure 3.1: The basic structure of wireless sensor networks

3.2 Simulation Model of Wireless Sensor Networks

To simulate wireless sensor networks, the simulation model is shown below:

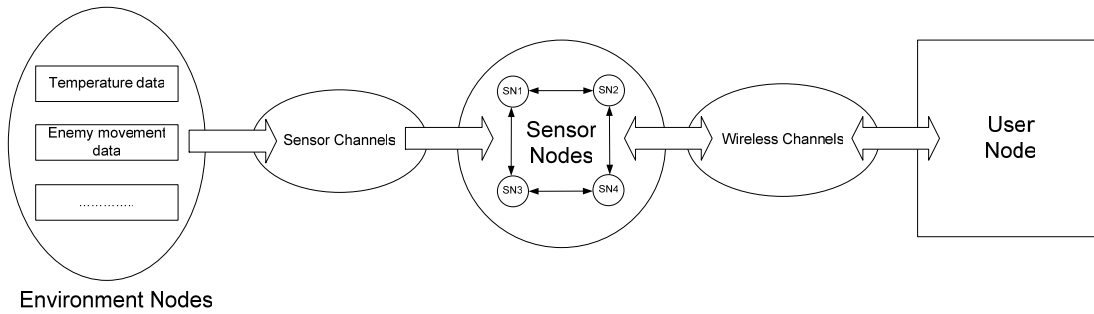


Figure 3.2: The simulation model of wireless sensor networks

The simulation model of wireless sensor networks is generally separated into four main models; environment, sensor node, user node and communication models.

Environment model: This model uses environment nodes to define the physical environment in the simulation. For example, in the enemy surveillance application, this model defines environment nodes moving randomly as enemy moving in the battle field or, in the environment temperature detection application, this model defines static environment nodes to generate the environment temperature in each area. The environment node model architecture is shown below:

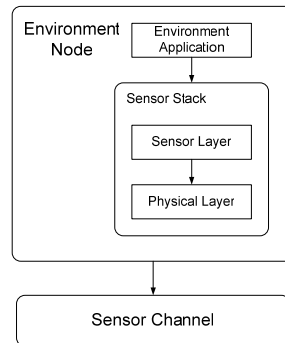


Figure 3.3: The environment node model architecture

The environment node consists of the application layer and the sensor protocol stack (sensor and physical layer). This node generates a physical data and broadcasts it down to the sensor channel.

Sensor node model: This model defines the conditions of sensor nodes such as node mobility, communication between nodes, routing protocol for reporting sensing data packet to the sink node etc. The sensor node model architecture is shown follow:

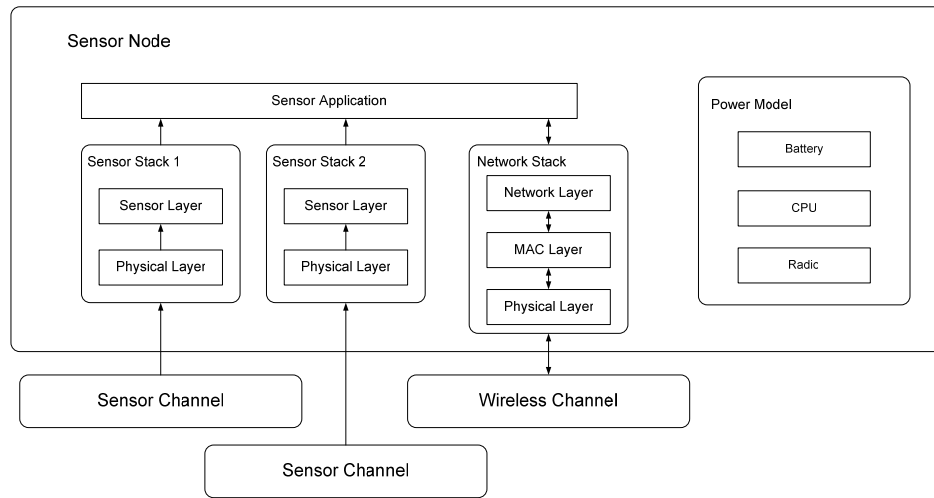


Figure 3.4: The sensor node model architecture

The sensor node model consists of one network protocol stack and one or more sensor protocol stacks depending on different types of environment targets. The task of the sensor protocol stack is to detect and process the detected data received from environment nodes on the sensor channel and forward them to the application layer. The application layer will process and transmit them to the user node in the form of sensor reports via the wireless channel. One important part of the sensor node besides these two types of protocol stack is a power model, which consists of an energy provider such as the battery and energy consumers such as CPU and Radio.

User node model: This model makes an interface between the sink node and users. Users use sensing data from the sink node to analyze their targets. The user node model architecture is shown below:

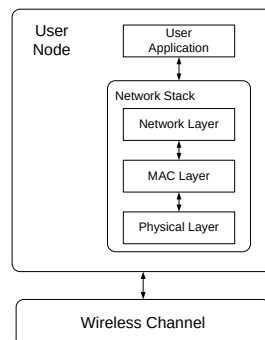


Figure 3.5: The user node model architecture

The user node, which consists of the application layer and the network protocol stack (network, MAC and physical layer), receives sensor reports and sends them up to the application layer.

Communication model: The wireless sensor network communication can be classified into three types.

Environment - Sensor Communication: This is a one-way-communication for broadcasting a physical environment data from the environment nodes to the sensor nodes. The communication channel between environment and sensor nodes is defined as a sensor channel.

Sensor - Sensor Communication: This is a two-way-communication between the sensor nodes. These traffics are used to collect signal-processing of sensor events in the network before they report them to the user node.

Sensor - User Communication: This is a two-way-communication the one from sensor to user is used to transmit sensor reports to the user, and the other, which is from user to sensor, is used to transmit user commands and queries to the network. The communication channel between sensor and user nodes is defined as a wireless channel.

3.3 Implementation Sensor Network in NS-2.28

As the network simulator (NS) does not support the wireless sensor network, the Naval Research Laboratory group [20] has extended some components from the wireless network in NS-2 to be used with the wireless sensor network. Unfortunately, this project was based on the network simulation version 2.1b9a (NS-2.1b9a) which is the old version of the NS and that cannot be used in the new version NS-2.28 which has more components to support the networks than NS-2.1b9a has. Our implementation of the sensor network in NS-2.28 is based on the Naval Research Laboratory group. The process of implementation is divided into two main processes; adding the wireless sensor network classes and modification codes in NS-2.28.

The overview of this implementation is as follows:

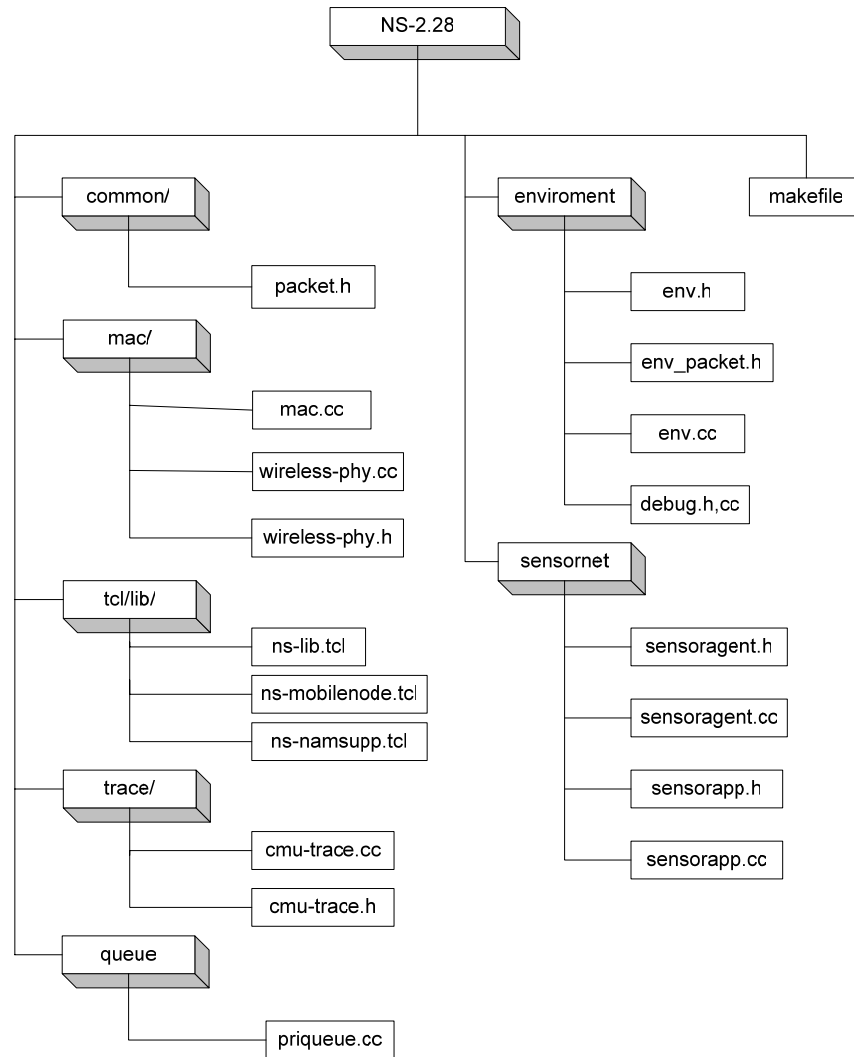


Figure 3.6: The implementation process of adding class (in right side) and modification files (in left side)

3.3.1 Wireless Sensor Network Classes

The wireless sensor networks are classified into three different types:

3.3.1.1 Environment Class

The environment class consists of environment nodes, environment agents, environment routing protocols, the pulse rate etc. The environment node needs to be attached to the environment agent to enable the node to communicate with the sensor channel. To broadcast the environment packet, the environment routing protocol and the pulse rate are required. The environment routing protocol is used to find the path and the pulse rate is used to indicate how often the environment node broadcasts the environment packet.

3.3.1.2 Sensor Agent Class

The sensor agent class consists of sensor nodes, sensor agents, UDP, TCP agent etc. The sensor node is attached to the sensor agent to communicate with the sensor channel for consuming the sensing packet. On the other hand, the sensor node is attached to a UDP or TCP agent to communicate with the wireless channel which receives the constructed packet sent down from the sensor application.

3.3.1.3 Sensor Application Class

The color and the rewriting functions are used in this class. The color function defines the different states of the sensor nodes by using different colors, and the rewriting function allows the other people to implement their own application.

3.3.2 Adding wireless sensor network classes

3.3.2.1 Environment Class

This module contains the new environment packet type, the environment agent and the environment packet emanating protocol. The environment agents send periodic pulses through a node's wireless interface, according to the period parameter specified in the Tcl script (for example: `[$node_(0) set ragent_] pulserate .5`), which have no auditory faculty, which means they don't listen and receive packets via attached sensor channel. To simplify the environment emanating routing protocol, we only broadcast the `Env_packet`. The main parts of the codes are:

Packet Types

`Env_packet.h` file contains all data structures, constants and macros related to these new packet types. The codes are:

```
1:  #define TEST_PHENOMENON 1
2:  #define CO 2
3:  #define HEAVY_GEO 3
4:  #define LIGHT_GEO 4
5:  #define SOUND 5
6:  #define HDR_ENV(p) ((struct hdr_env_main*)hdr_env::access(p))
7:  #define ENV_PAYLOAD(p, offset) ((struct payload*)hdr_env::access(p, offset))
8:  struct hdr_env {
9:      static int offset_;
10:     inline static int& offset() { return offset_; }
11:     inline static hdr_env* access(const Packet* p) {
12:         return (hdr_env*) p->access(offset_);
13:     }
14:     inline static hdr_env* access(const Packet* p, const int loffset) {
15:         return (hdr_env*) p->access(offset_+loffset);
16:     }
```

```

17: };
18: struct hdr_env_main {
19:     u_int16_t pck_length;
20:     u_int16_t reserved;
21:     inline int size() {
22:         int sz = pck_length;
23:         assert(sz>0);
24:         return sz;
25:     }
26: };
27: struct payload {
28:     nsaddr_t msg_oAddr;
29:     u_int16_t msg_size;
30:     u_int16_t msg_seqno;
31:     u_int16_t environment;
32:     inline int size() {
33:         int sz = msg_size;
34:         assert (sz >= 0);
35:         return sz;
36:     }
37: };

```

Lines 1 to 5 define the different types of the environment packets. Lines 6 to 37 declare the general header with functions for all environment packet structures. Lines 28 to 31 are the raw attributes of new packets.

- msg_Addr – describes originating address of broadcast.
- msg_seqno – describes sequence number of this packet.
- msg_size – describes the size of the payload
- environment – indicates which types of environment packets have been transmitted as they are defined in lines 1 to 5

The environment agent and the environment packet emanating routing protocols are implemented in the main env.cc file

TCL Hook

Tcl hook is used to bind the new environment packet and the new environment agent to Tcl and let env_packet and env_agent be instantiated from Tcl.

```

1: static class ENVHeaderClass : public PacketHeaderClass {
2:     public:
3:     ENVHeaderClass() : PacketHeaderClass("PacketHeader/ENV", //0) {
4:         sizeof(hdr_env))    bind_offset(&hdr_env::offset_);
5:     }
6: } class_rtProtoENV_hdr;
7: static class ENVclass : public TclClass {
8:     public:

```

```
9:     ENVclass() : TclClass("Agent/ENV") {}
10:    TclObject* create(int argc, const char*const* argv) {
11:    assert(argc == 5);
12:    return (new ENV((nsaddr_t) atoi(argv[4])));
13:    }
14: } class_rtProtoENV;
```

Command()

It consists of the implementation of the `command()` method that the environment agent inherits from the agent. It seems little more complicated. The codes are shown in [Appendix B1](#)

```
ENV::command(int argc, const char*const* argv)
```

This is the normal format of the command functions. `argv[0]` contains the name of the method being invoked, `argv[1]` is the requested operation, and `argv[2..argc-1]` is the rest of the arguments which were passed. Within this function, we code some mandatory operations as well as any other operations which we want to make accessible from Tcl. The following codes show the example when we configure the pulse rate in the simulation script and how the command function fetches it.

```
1:     else if(argc == 3) {
2:         if(strcmp(argv[1], "pulserate") == 0) {
3:             Hello_Interval = atof(argv[2]);
4:             return TCL_OK;
5:         }
```

Emanate()

The `ENV::emanate()` function, which is implemented for the environment packet broadcast in the attached environment channel, builds and sends off a hello message. The codes are shown in [Appendix B2](#)

In the environment module, there are two parameters, which can be configured in the simulation script. One is the pulse rate that actually controls how often the environment packet emanates, and the other is the environment type (Carbon Monoxide, heavy seismic activity, sound etc)

3.3.2.2 Sensor Class

This module contains the new sensor agent and sensor application. The Agent is defined in ns2 manuals as endpoints where network layer packets are constructed or consumed. At least two agents are attached to the sensor node, a sensor agent is attached to the sensor channel for consuming the environment packet and a UDP or TCP agent is attached to the wireless channel for constructing packet, and sending it down to the sensor application. The sensor agent, which acts as conduit through which environment packets are received and forwarded to the sensor applications, is implemented in the `sensoragent.h,cc`. The main parts of the sensor agent are:

TCL hook

Tcl hook is used to bind our new agent to Tcl to let the sensor agent be instantiated from the Tcl. To do this, we must inherit from the class `TclClass` as shown in the following code

```
1: static class SensorClass : public TclClass {
2: public:
3:     SensorClass() : TclClass("Agent/SensorAgent") {}
4:     TclObject* create(int, const char*const*) {
5:         return (new SensorAgent());
6:     }
7: :} class_sensor;
```

In line 3, the class constructor merely calls the base class with the string `Agent/sensorAgent` as an argument, which represents the class hierarchy of this agent in a textual manner. In lines 4 to 6, a function called `create()` is implemented to return a new `SensorAgent` instance as a `TclObject`.

Recv()

This `recv()` is invoked whenever the sensor agent receives an environment packet. Every packet has a common header called `hdr_cmh` defined in `common/packet.h`. To access this header, a macro like the one defined before is used for our own packet types, and we use it at line 3. In lines 2 to 7, they check the sensor application whether it is attached to the agent, if the sensor application is attached, they will pass the environment packet to it.

```
1: void SensorAgent::recv(Packet* pkt, Handler*) {
2:     if (sensor_app) {
3:         hdr_cmh* ch = hdr_cmh::access(pkt);
4:         sensor_app->process_sensor_data(ch->size(), pkt);
5:     }
6: }
```

Attach_sensor_app()

This function received from the node's sensor application once the application receives a start command. In line 3, it sets the `sensor_app` variable to the `sensor_app` parameter because this parameter hides class variables.

```
1: void SensorAgent::attach_sensor_app(SensorApp* sensor_app_param)
2: {
3:     sensor_app = sensor_app_param;
4: }
```

3.3.2.3 Sensor Application Class

The sensor application generates the sensor report to show when the corresponding sensor node detects the environment packet. When the sensor agent receives the environment packet and passes it to the sensor application, it activates an “alarm” public variable in the NAM visualization tools, changes the node color to red, and, at the same time, the sensor application sends a sensor report of MMSG_SIZE bytes to the sink node via a UDP or TCP connection once per TRANSMIT_FREQ seconds. In the timeout period, when the sensor node does not receive an environment packet, the state of node changes to the silent state. In NAM, the node color is changed to green. This application implemented in the sensorapp.h.cc.

TCL Hook

Like the sensor agent, Tcl hook is used to generate a new instance as a TCL object like Application/SensorApp.

Start()

This function tells the sensor agent that the node has a sensor application attached and a pointer passed to this sensor application, so the sensor agent can notify the sensor application when an environment packet is received.

```
1: void SensorApp::start(SensorAgent* sensor_agent) {
2:     sensor_agent_ptr = sensor_agent;
3:     sensor_agent_ptr->attach_sensor_app(this);
4:     STARTED = TRUE;
5:     alarm = DEACTIVATED;
6:     change_color("green");
7: }
```

Lines 2 to 4 tell the sensor agent that a new sensor application is attached. Lines 5 to 6 initiate the parameters for change_color function and set the sensor node color as green which means the node is in the silent state.

change_color()

This function is used to change the color of the Tcl object nodes when the node switches the status between activity and silent.

```
1: int SensorApp::change_color(char *color) {
2:     if (DISABLE_COLORS)
3:         return 0;
4:     if (strcmp (newcolor, color) != 0 && STARTED) {
5:         Tcl& tcl = Tcl::instance();
6:         tcl.evalf("%s set node_", sensor_agent_ptr->name());
7:         const char *node_object = tcl.result();
8:         Tcl::instance().evalf("%s color %s", node_object, color);
```

```
9:      newcolor = color;      // make pointers equal
10:    }
11:    return 0;
12:  }
```

In lines 2 and 3, if the node color is desired to illustrate energy levels instead of sensor activity status, then it will use `DISABLE_COLORS` to turnoff the color change function in the sensor application. In lines 4 to 10, these codes are used to change the node color during the simulation and report it to the NAM tracing file.

Process_sensor_data()

The `SensorAgent` function calls this function after it detects some environment packet and periodically sends `MESG_SIZE` bytes of data to the sink node.

```
1:  void SensorApp::process_sensor_data(int size, Packet* env_pkt) {
2:      change_color("red");
3:      if (alarm == DEACTIVATED) {
4:          agent->send(MESG_SIZE);
5:          send_timer.resched(TRANSMIT_FREQ);
6:      }
7:      alarm = ACTIVATED;
8:      silence_timer.resched(SILENT_ENV);
9:  }
```

Lines 3 to 6 illustrate when the sensor node is active, the sensor node only forwards the packet.

Timer

The timer expires after `TRANSMIT_FREQ` seconds, since this sensor node transmits the last sensor data report to the sink. This timer prevents nodes from transmitting a sensor data report every time they hear an environment packet (which would be very bad in the simulation where there are a lot of environment nodes.) The timers have two functions:

SendTimer::expire(Event *e)

When the node alarm is activated, it sends the `MESG_SIZE` bytes message every `TRANSMIT_FREQ` seconds to the sink node via the attached transport agent (udp, tcp, etc).

SilenceTimer::expire(Event *e)

This function defines what should happen when the sensor is no longer detecting the environment packet. Such as turning off the node alarm, changing the node color to the silent color, and cancelling the sent timer.

In the sensor module, there are configurable parameters which can setup in the simulation such as `SILENT_ENV`, `DISABLE_COLORS`, `MESG_SIZE`, and `TRANSMIT_FREQ`.

3.3.3 Modification in NS-2.28

As shown in Fig 4.5, the modification details of files in NS-2.28 are explained follow:

.common/packet.h

In the NS, each packet has a unique type to associate with a specific protocol such as UDP, MAC, AODV, TCP etc. In Env class, a new protocol is created to broadcast the `env_packet`, so it is required to add the new packet type to the `packet.h` header file. To add this new packet type, the following command lines are put in the `packet.h` file.

```
add: "#define HDR_ENV(p) (hdr_env::access(p))"  
      "PT_ENV" to enum packet_t type  
      "name_[PT_ENV]= "ENV";" to p_info()
```

./MAC/ mac.cc and wireless-phy.cc,h

In the wireless sensor network, the sensor node contain at least two interface types; Sensor and environment interfaces. So, the modification `mac.cc` file can avoid the interface counting problem.

NS-2 contains the energy model for wireless nodes which can describe the power consumption of the wireless node in different states. The model includes attributes for specifying the power requirements of transmitting packets, receiving packets, and the idle standing. The sensing environment may consume another rate of power, so we must add these power consumption types in `wireless-phy.cc,h` files.

To modify these two files, the following codes are added.

For `mac.cc`,

```
Add: "} else if (strcmp(argv[1], "adjust_index") == 0) {  
      MacIndex--;          return (TCL_OK);"
```

These codes are added to prevent the `MacIndex` from the counting mistake which increases double after adding multiple interfaces to a sensor node.

For `wireless-phy.h`

```
Add: "double Ps_consume_;" power consumption for sensing environment in sensor node (W)  
      "fprintf(stderr, "wireless-phy.h: downtarget_ == %d\n", downtarget_);"
```

.tcl/lib/ns-lib.tcl

This Tcl script file defines the node configurations, which are specified in the NS-2 Tcl simulation script. In the extension, we add two new node types; environment and sensor nodes; therefore, we must add the description details of the new type nodes in this Tcl file.

In the switch-exact \$routingAgent_ section, the under-mentioned codes are added.

```
Add: ENV {
    set ragent [$self create-env-agent $node]
}
"—ENVchannel" to the environment node configuration
"—sensePower (in W)" to sensor node power discription
```

Add the following functions to generate a new channel and sensor power objects:

```
Simulator instproc ENVchannel {val} {$self set ENVchannel_ $val}
Simulator instproc ENVmacType {val} {$self set ENVmacType_ $val}
Simulator instproc sensePower {val} { $self set sensePower_ $val }
```

Add the following functions to attach the agent to the environment node:

```
Simulator instproc create-env-agent { node } {
# Creating ENV routing agent
set ragent [new Agent/ENV [$node id]]
$self at 0.0 "$ragment start"      ;# start BEACON/HELLO Messages
$node set ragent_ $ragment
return $ragment
}
```

.tcl/lib/ns-mobilenode.tcl

This Tcl script file, which defines the wireless network channel description, supports the multi-channel wireless network. In the extension, the sensor nodes have at least two interfaces, one for the wireless channel and another for the sensor channel. The modification implements this kind of “multi-homed” capability in ns-mobilenode.tcl. The modified code of ns-mobilenode.tcl is shown in [Appendix B3](#).

.tcl/lib/ns-namsupp.tcl

This tcl file contains nam_api to support NS-2, where the old nam_api (set_rate) works, but it is quite obscure. The following codes are added in this file.

```
Add: "set r [time_parse $rate]"
"$self puts-nam-config "v -t [$self now] set_rate_ext $r 1"
```

./trace/cmu-trace.h,cc

The CMUTrace class is used to export the important part of a packet to a simulation trace file-name.tr which is defined in the simulation Tcl script. Since there is a new packet implanted to env_packet, we need to describe this new packet format in this class. The modified code of cmu-trace.h is added the under-mentioned code, while the modified code of cmu-trace.cc is shown in [Appendix B4](#).

Add: "void format_env (Packet *p, int offset);" as a private function

./queue/priqueue.cc

NS-2 contains different queue algorithms. Since there is a new Env type packet implemented, it needs to choose a specific queue algorithm to associate with. To modify this code, the following code is added:

Add: "case PT_ENV:" to the PriQueue::recv function

Makefile.in

Makefile is used to compile the OTcl object in NS-2. When new modules or components are added, the corresponded object is needed to be added to the Makefile to build it. In this extension, we need to rebuild all these modified objects in the base NS-2 object and build new objects such as env.o, sensoragent.o and sensorapp.o. The following codes are added in this file:

Add: "Enviroment/evi.o sensornets/sensoragent.o sensornets/sensorapp.o \" to generate the Tcl object.

3.4 Example of Using the Simulator to Test Routing Protocols

After implementing NS-2.28 based simulator, we use this simulator to test the DSDV and AODV routing protocols. In this part, the specific models, the experiment scenario and the results are described.

3.4.1 Physical simulation model

In our experiment, we assume that the environment nodes detect the environment target in the wide area space, where we use a two-dimensional model (x-y plane) which is a square space of 1000m x 1000m. We use three types of nodes; user, sensor and environment nodes. Moreover, ten environment nodes moving randomly in this square space are used different numbers of sensor nodes which are 49, 100, 225 and 400 fixing in this space. The environment nodes, which detect the environment target, generate data packets sent to the sensor nodes, and then the sensor nodes report these packets to the user node. In fact, the user node, which is used for collection the

sensor report, is the destination of all data packets. The examples of simulation models are shown below.

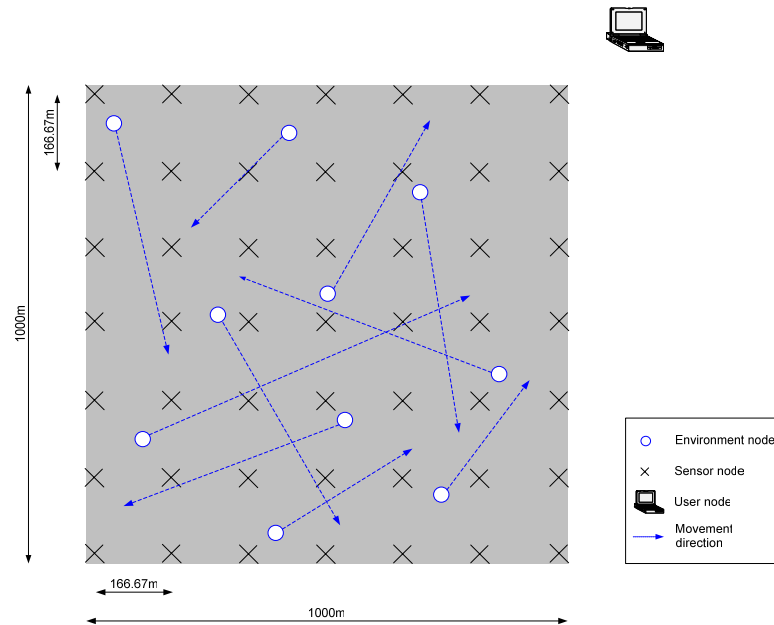


Figure 3.7: The simulation model of the 49 sensor nodes in 1000m x 1000m with 10 random movement environment nodes.

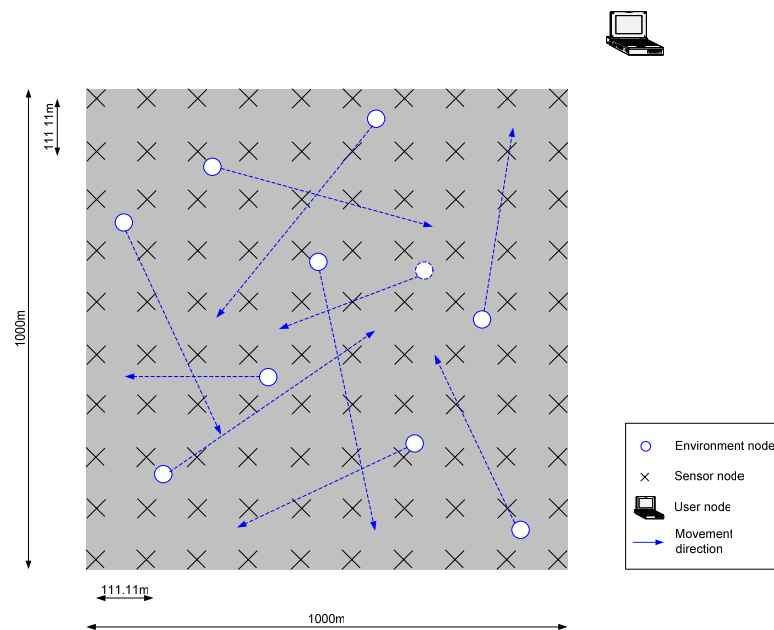


Figure 3.8: The simulation model of the 100 sensor nodes in 1000m x 1000m with 10 random movement environment nodes.

3.4.2 Environment node model

Once the environment node, which is used to detect the environment target, detects the target, it generates a data packet and sends it to the sensor nodes. The environment node model consists of:

- Link layer
- MAC layer with the 802.11 type
- Interface queue with DropTail/PriQueue
- Interface queue length with maximum 50 packets in the interface queue
- Antenna with Omni-directed Antenna
- Propagation model with TwoRayGround
- Network interface with WirelessPhy
- Channel with SENSOR channel
- Routing protocol with Env routing protocol

3.4.3 Sensor node model

The sensor node receives the detected data from the environment nodes and reports them to the user node. The sensor model consists of:

- Link layer
- MAC layer with the 802.11 type
- Interface queue with the DropTail/PriQueue
- Interface queue length with maximum 50 packets in the interface queue
- Antenna with the Omni-directed Antenna
- Propagation model with the TwoRayGround
- Network interface with the WirelessPhy
- Channel with the SENSOR and the WIRELESS channels
- Routing protocol with DSDV or AODV

3.4.4 User node model

The user node collects all the sensor reports from the sensor nodes. The user node model consists of:

- Link layer
- MAC layer with the 802.11 type
- Interface queue with the DropTail/PriQueue
- Interface queue length with maximum 50 packets in the interface queue
- Antenna with the Omni-directed Antenna
- Propagation model with the TwoRayGround
- Network interface with the WirelessPhy
- Channel with the Wireless channel
- Routing protocol with DSDV or AODV

3.4.5 Communication model

To receive the detected data from the environment nodes via the sensor channels, the sensor nodes need to attach to the sensor agents to communicate with sensor channels. For the communication between the sensor nodes and the user node via wireless channel, the sensor nodes also need to attach with the UDP agents; whereas, the user node need to attach to the sink agent.

3.4.6 Mobility model

As mentioned above, the sensor and the user nodes are fixed. We create the random movement only for the environment nodes by using the Setdest function in NS-2. The starting positions of the environment nodes are randomly initiated in a square area of 1000m x 1000m and they move randomly to their random destinations with the speed between 0 - 20 m/s. When they reach their destination, they will move to the next random destination until the simulation time is expired.

3.4.7 Experiment scenario

In this experiment, the four and ten environment nodes are used in the model discussed above. Testing by using ten environment nodes, the sensor and the user nodes use the AODV routing protocol. The environment nodes are set to broadcast the packets once they move into each sub-space. The time to run the simulation is 50 seconds. After finishing the AODV routing protocol, the sensor and user nodes are changed to use the DSDV routing protocol. The simulation also runs in the same condition. For testing with four environment nodes, the AODV and the DSDV are also used in the same condition.

The interesting parameters in this experiment are the packet delivery ratio and the average end-to-end packet delivery time. The formulae for the calculation of these two parameters are

$$\text{Packet delivery ratio} = \frac{\text{Received_Packets}}{\text{Sent_Packets}} \times 100$$

$$\text{Average end-to-end packet delivery time} = \frac{\sum (Tr - Ts)}{\text{Number.of.Packets}}$$

Where Tr = The receive time

Ts = The send time

3.4.8 Results

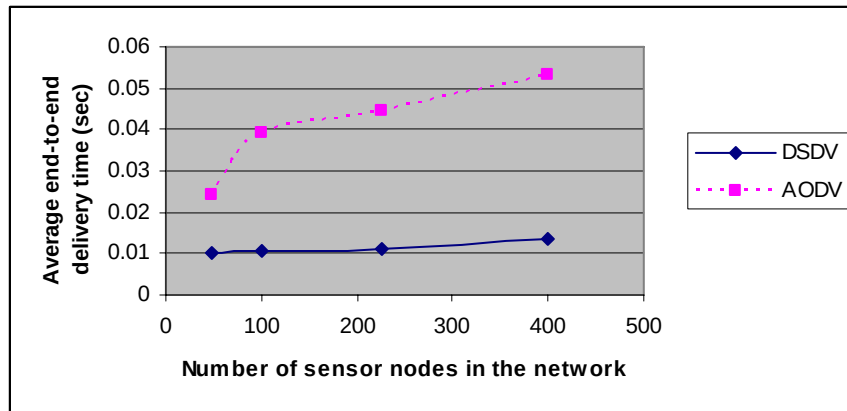


Figure 3.9: Average end-to-end delivery time of DSDV and AODV with different number of sensor nodes

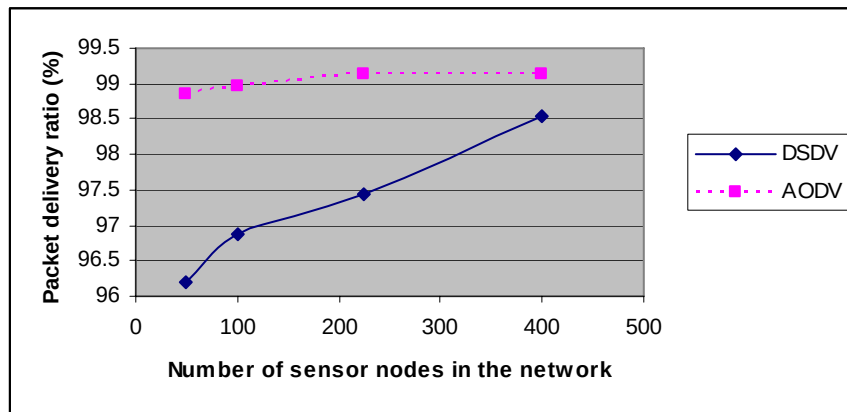


Figure 3.10: Packet delivery ratio of DSDV and AODV with different number of sensor nodes

With the testing condition as mentioned above, DSDV has a better performance in the average end-to-end delivery time while AODV has a better performance in the packet delivery ratio. The simulation in this part is only the example of using the implemented protocol to test routing protocols. These results are not the conclusion of these two protocol comparison since we neglect some different conditions of these protocols such as the speed of node mobility and the size of sensor nodes should be used more different sizes.

4 Extension of Simulator for Specific Sensor Network Applications

After extending NS-2.28 to support the wireless sensor network simulation, the simulator is extended for use in a specific sensor network. In this chapter, the specific sensor network application models and the method to extend the NS-2 based simulator for specific sensor network applications are discussed.

4.1 Specific Sensor Network Application Models

This part identifies the specific sensor network application models. The sensor network applications such as enemy surveillance, chemical gas cloud detection, and detection of environment temperature are discussed.

4.1.1 Enemy surveillance application model

The enemy surveillance is one important application using in the military application. An amount of sensor nodes are deployed in the wide environment area which may be attacked from enemies. These sensor nodes are attached with ground vibration sensor. When enemies move into the sensor area, these sensor nodes will detect the ground vibration and generate the detected data packet sending to the sink node which is interfaced with the commander. The enemy surveillance application model is shown below.

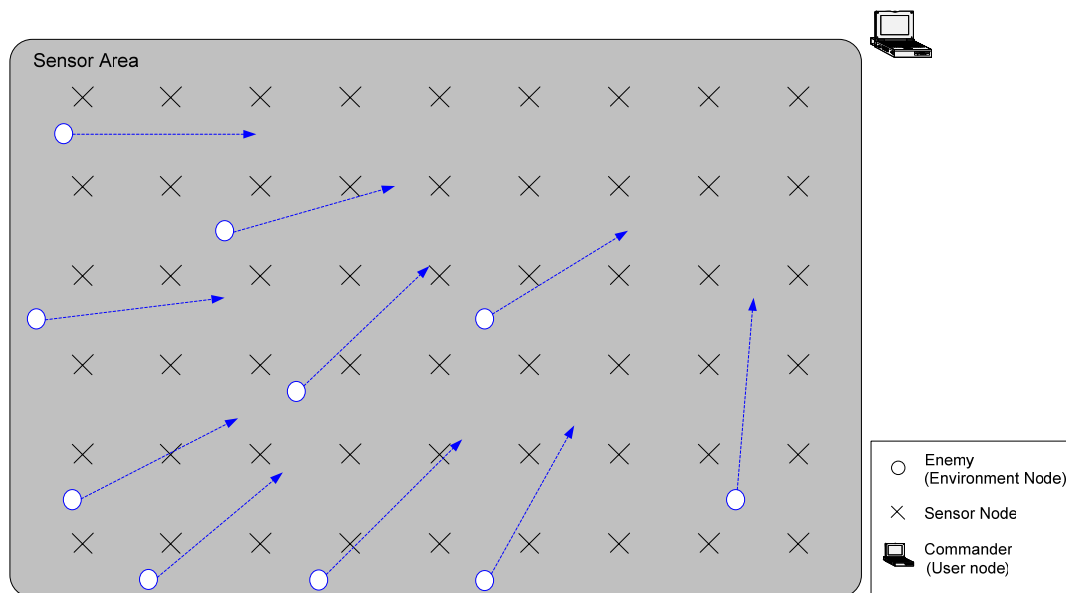


Figure 4.1: The sensor network model used for the enemy surveillance application

In the simulation model, enemies are assumed as environment nodes moving into the observe sensor area. The sensor nodes are coordinated the fix position in the sensor area and are given the identify numbers from the commander. The behaviour that the sensor nodes detect the broadcasted “ping” from environment nodes is assume as the sensor nodes detect the ground

vibration from enemies moving in the sensor area. Once these sensor nodes detect “ping”, they generate data packets and send them to the user node. When the user node receives these detected data, it analyzes the position where enemies attack by using the identity numbers of sensor nodes which send these detected data to it.

4.1.2 Chemical gas cloud detection application model

The chemical gas cloud detection application is very important for the industries which use the chemical gas as one of power sources. Since some chemical gas is hazardous for people, the sensor surveillance is needed to detect this gas before it emanates to the wide area. The chemical gas cloud detection model application is shown below.

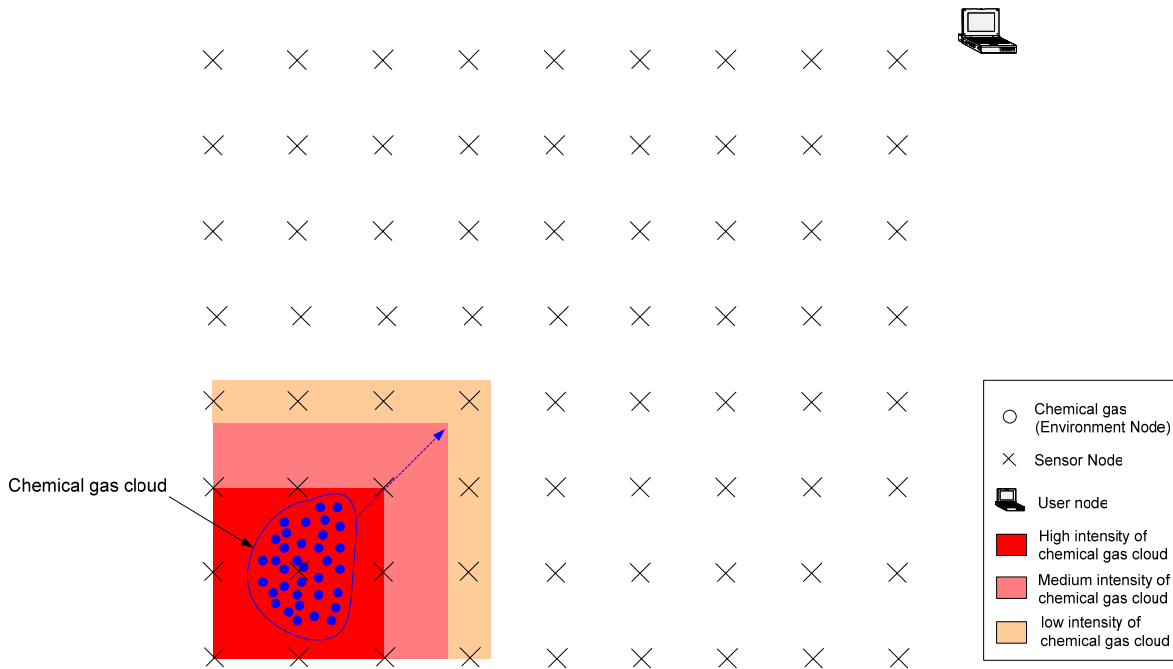


Figure 4.2: The sensor network model used for the chemical gas cloud detection application

This model assumes the group of environment nodes as the chemical gas cloud. These environment nodes move together in the same direction. The sensor nodes in this application are fixed in the position given by users. These sensor nodes use the threshold values to define the intensity of the chemical gas cloud. For example, the threshold value “1” is used for a high intensity, “0.5” is used for a medium intensity, and “0.05” is used for a low intensity. When these sensor nodes detect “ping” which is broadcasted by the environment node (the behaviour that sensor nodes detect “ping” is assume as these sensor nodes detect the chemical gas cloud), they compare the signal power of “ping” with threshold values to identify the intensity of the chemical gas cloud. The closer sensor nodes to environment nodes give the higher signal power and higher intensity of the chemical gas cloud. After sensor nodes define the intensity level of the chemical gas cloud, they generate data packets and send them to the user node. From these data, users know where the chemical gas cloud is, and intensity of the chemical gas cloud in that area.

4.1.3 Environment temperature detection application model

The simple environment temperature detection application model is shown below:

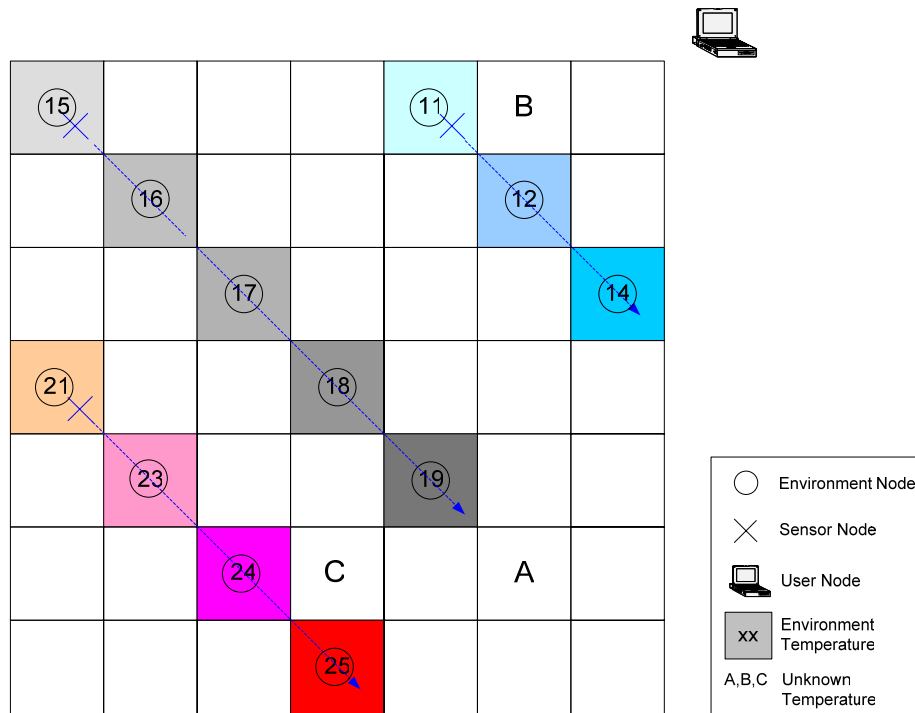


Figure 4.3: The sensor network model used for the environment temperature detection application

This model defines static environment nodes to broadcast data packets which include the temperature in their areas while sensor nodes move in the same direction as shown in figure 4.3 to detect each temperature area. In this model, the unknown temperature A, B and C can be approximated and calculated from the neighbor area temperature which is detected by sensor nodes. For example, the temperature A can be forecasted from the earlier detected temperature where A could be 20 degrees, the temperature B is the average of 11 and 12 which is 11.5 degrees and temperature C can be calculated by finding the average temperature of the closed area temperature (19, 24 and 25) which is 22.67 degrees.

4.2 Extension method of the NS-2 Based Simulator for Specific Applications

In this part, two main extensions of the NS-2 based simulator for specific sensor network application are discussed.

4.2.1 Extension of environment module

To simulate sensor network applications, the environment module is needed to extend to support specific application models. The main task of this module is to provide the environment data for the simulation. In the enemy surveillance and the chemical gas cloud detection applications, this module provides the amount of environment nodes (assumed as enemies and chemical gas) and the mobility of environment nodes, while, in the environment temperature detection application, this module provides the temperature in each area to environment nodes.

The extension of this module can be written in C++ and then interface it with NS-2. The information how to interface a new module with NS-2 can be found in [21].

4.2.2 Extension of real data transmission module

Since we need to use the real user data, the main problem is how to create the real user data and transmit them. In NS-2, all applications are described as virtual applications, which do not actually transfer their own data in the simulator including the size and the time when data are transferred. As we shown in here, when we do some specific sensor network simulations which are not only the network behaviours that we want to know but also how the functional of the specific sensor network is, we need to transfer the real application level's data. The method to implement these modules in the extension of NS-2.28 is a big challenge for us, who have the limitation of programming experience and time to implement. Below are some ideas and general steps to implement these modules.

In order to transmit the application-level data in NS-2, we need a uniform structure to pass these data among applications, and send them down from applications to transport agents. To do this, three major components which are a representation of a uniform application-level data unit (ADU), a common interface to pass data between applications, and two mechanisms to pass data between applications and transport agents such as UDP agent and TCP agent are needed. The basic components that can be used to generate the user data are:

4.2.2.1 Application-level Data Unit (ADU)

The function of an ADU is similar to a `Packet` unit function which is needed to pack user data including the user data area of an NS packet by an Agent into an array. In the current NS-2, the agent is not supported by current Agents, so we must derive new agents to accept the user data from applications (UDP), or use an agent wrapper (TCP).

Compared with `Packet`, ADU provides this functionality in a different way. In `Packet`, a common area is allocated for all packet headers; an offset is used to access different headers in this area. In ADU, this is not applicable because some ADUs allocate their space dynamically according to the availability of the user data. The below codes show the abstract base class of all ADU:

```
1: class AppData {
2:     private:
3:         AppDataType type_; // ADU type
```

```
4:     public:
5:         struct hdr {
6:             AppDataType type_;
7:         };
8:     public:
9:         AppData(char* b) {
10:             assert(b != NULL);
11:             type_ = ((hdr *)b)->type_;
12:         }
13:     virtual void pack(char* buf) const;
14: }
```

In line13, `pack(char* buf)` is used to write an `AppData` object into an array, and `AppData(char* b)` is used to build a new `AppData` from a serialized copy of the object in an array.

4.2.2.2 A common interface to pass data between applications

The base classes of `Application` and `Process` allow applications to pass data or request data between each other. The process which enables `Application` to link `Process` is:

```
1: class Process {
2:     public:
3:     Process() : target_(0) {}
4:     inline Process*& target() { return target_; }
5:     virtual void get_data (int& size, char* req_data = 0)
6:     virtual void process_data(int size, char* data) = 0;
7:     virtual void send_data(int size, char* data = 0);
8:     protected:
9:     Process* target_;
10: };
```

Lines5-6 are three public methods provided by the base class process to deal with the user data, the method `get_data` is used to request data from the previous application in the chain, `send_data` is used to send data to the next application in the chain, and `process_data` is used to process incoming data

4.2.2.3 Two mechanisms to pass data between applications and transport agents

There is no supported `Agent` class to transmit the user data in the current NS-2.28, but there are two ways to transmit a serialized ADU through the transport agents. First, for UDP agent, we can derive a new agent from `UDP` class and add a new method `send(int nbytes, char *userdata)` to pass the user data from `Application` to `Agent`. To pass the data from `Agent` to `Application` is tricky because each agent has a pointer to its attached application, then it dynamically casts this pointer to an `AppConnector` and calls `AppConnector::process_data()`. Second for `TCP` agent, where the trickery is more than doing that over `UDP`, because `TCP`'s reassembly queue is only

available for FullTcp. But the problem can be solved by abstracting a TCP connection as a FIFO pipe.

Therefore, for the sensor network application we need to drive a new ADU which calls AensorAppData from the base class. The new ADU class should contain a general data structure for all sensor application data, which are not only a unique data structure of the specific sensor application like the temperature data structure. As we think the normal sensing information like the temperature information is short, we use the UDP agent to send this information. This process is defined below:

```
1:  class SensorAppAgent: public Agent {
2:      public:
3:      SensorAppAgent();
4:      virtual void recv(Packet *, Handler *);
5:      virtual void send(int realsize, AppData* data);
6:      protected:
7:      int off_inv_;
8:  };
```

In lines4-5, the method `recv(Packet*, Handler*)` is overridden to extract the user data, and a new `send(int, AppData*)` is provided to include the user data in packets. In `send()`, the following code can be used to write the user data from `AppData` to the user data area in a packet:

```
1:  Packet *pkt = allocpkt(data->size());
2:  hdr_inval *ih = (hdr_inval *)pkt->access(off_inv_);
3:  ih->size() = data->size();
4:  char *p = (char *)pkt->accessdata();
5:  data->pack(p);
```

In `recv()`, the following code can be used to read the user data from the packet and to deliver it to the attached sensor application:

```
1:  hdr_inval *ih = (hdr_inval *)pkt->access(off_inv_);
2:  ((SensorApp*)app_)->process_data(ih->size(), (char*)pkt->accessdata());
3:  Packet::free(pkt);
```

As described above, the basic idea to generate and transmit the user data in the based NS-2 sensor network simulators is shown below:

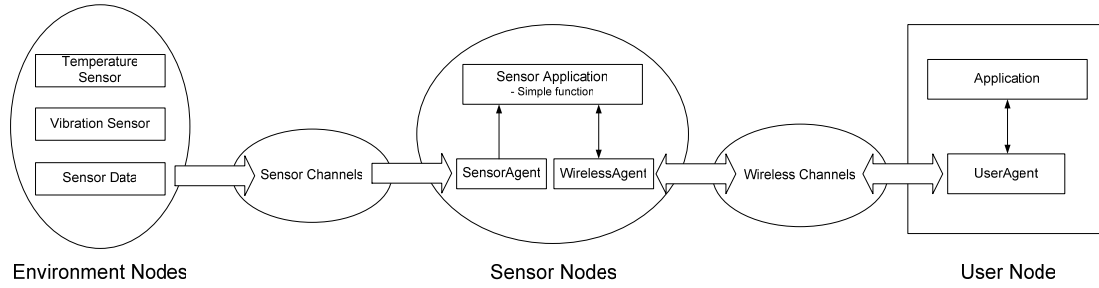


Figure 4.4: The passing data process from the environment nodes to the user node

Figure 4.4 shows how the sensor data pass the whole sensor network and finally receive and deal with the user node. The environment node generates different kinds of sensor data to send them to the attached sensor channel, and when the sensor nodes detect the environment nodes, the sensor node agents hand the received sensor data to the sensor application which is attached to the sensor nodes. After the sensor application does some basic functions with the sensor data, the sensor data will be sent to the user node via the attached wireless channel. Finally, the sensor data will be reported and analyzed by the user application.

The methods how to implement this are shown below:

Step 1 Generate sensor data in the environment module

In the environment module, we need to create a new class which can be called `sensorappdata`, which is extended from the `appdata`. The `sensorappdata` class is needed to contain the sensor data structure for general environment data and some methods like pointing the user data, inserting the sensor data in array etc. Required by different sensor application, we need to generate a new class `tempappdata` which is inherited from the `sensorappdata` to specify the sensor data.

Step 2 Transmit sensor data in the sensor module

Since ADU is similar to a `Packet` unit but has something different as we discussed above, thus, in the extension, we need to modify the `sensoragent` to support the transmission of the sensor data from the sensor agent to the sensor application which is attached in the sensor node. To communicate with the user node via wireless channels, a new agent `wirelessagent` which supports to transmit the sensor data is needed. Depending on the service required, we can inherit this new agent from `UDP` or `TCP`. We need to modify the existing `sensorapp` class to support the sensor data and basic functions to deal with these data. We can rewrite the `process` and `precess_data` to pass the sensor data between sensor applications. For the sensor node, we also give the basic functions to deal with the sensor data such as `filter()` and `compare()` which required to process these sensor data and send them down to the `wirelessagent`.

Step 3 Deal data in the user module

User module contains a new user agent `useragent` which is inherited from `UDP` or `TCP` basic agent to support the sensor data and a new application `userapp`. This application executes the

main function of the sensor network like measuring the environment temperature as it was discussed above.

Unfortunately, since the time is limited, we can only propose the sensor network application models and the methods how to extend the environment and real data transmission modules, which can be carried out in the future.

5 Conclusion

In this work, the structure of wireless sensor networks is defined, and then NS-2.28 is implemented by adding three sensor network classes; environment, sensor agent and sensor application classes and modifying the existing classes in NS-2.28 to support the wireless sensor network. Three different types of mobile nodes, namely environment, sensor, and user nodes, are defined in the wireless sensor network. The environment class is added to NS-2.28 for the environment node to generate the periodic pulses and send the packets to the sensor node, the sensor agent class is added for using the agent to communicate with environment and user nodes, and the sensor application class is added for the color and rewriting functions.

After implementing the NS-2 based simulator, the extension of simulator for specific sensor network application is discussed. The enemy surveillance, the chemical gas cloud detection and the environment temperature detection models are presented. Finally, the extension method of the NS-2 based simulator for specific applications are proposed. This method includes the extension of environment and real data transmission modules. Unfortunately, since the time is limited and is much spent in finding this extension method, the extension of this simulator can be done in the future.

5.1 Discussion

Since we are very new in the fields of wireless sensor networks and the network simulation, we spent a lot of time to learn how to write OTcl script in NS-2 and solve some NS-2 problems such as installation problems. Valuable information was obtained from [22]. However, we have some comments on NS-2 and the sensor network simulators.

Although NS-2 is the most widely used for the network simulator, it does not support directly the wireless sensor network, and to use this simulator with the sensor networks, some implemented modules are needed. Unfortunately, these implemented modules can be used only for the version of NS-2 that users implement, but they can not be reused in the different versions. Furthermore, some applications in the wireless sensor networks require some new modules for generating real data packets or new routing protocols. However, NS-2 is an object-oriented design which introduces much unnecessary interdependence between modules. This interdependence causes the implementation of new modules extremely difficult for users who do not have many experiences in the simulator. Moreover, since a sensor network needs to be tailored for a particular application with specific features, NS-2 has no such protocols or algorithms support to do that.

To simulate the wireless sensor networks, using the simulators which directly support these sensor networks should be better. There are large numbers of directly supportive sensor network simulators as we mentioned in chapter 2.3. SENS whose its modules are programmed in C++ and TOSSIM whose TinyOS codes are used support the high level sensor network simulation. But TOSSIM is not sufficient for the low level protocol such as MAC. OPNeT++ is also one option for the high level sensor network simulation. The simple modules of OPNeT++ are programmed in C++ while its compound modules are programmed in a high-level language (NED). Although

J-Sim provides supporting target, sensor and sink nodes, sensor channels and wireless communication channels, its use of Java as the simulating language is inevitably sacrificing the efficiency of the simulation. As the packet formats, energy models and MAC protocols are not representative of those used in wireless sensor networks, GloMoSim may not be a good option for the sensor network simulator. For SensorSim, it is unfortunately no longer developed and is not available.

5.2 Future Work

The extension of this simulator for specific applications such the applications mentioned in chapter 4 can be done by using our proposed extension method. While programming the software for the user to analyze the received data for its target is one of the interesting works. For some specific applications, the implementation of a new routing protocol is needed. The information concerning the implementation of a new protocol can be found in [\[23\]](#).

Furthermore, one of the challenging works is to make a new sensor network simulator which supports the simulation in the high level application. This new simulator should also be possible to apply for other applications in easy ways of implementations.

6 References

- [1] The Network Simulator –ns-2, Information Science Institute, USC Viterbi School of Engineering, <http://www.isi.edu/nsnam/ns> [accessed 25 May 2005].
- [2] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, E. Cayirci, “A survey on sensor networks”, IEEE Communications Magazine, August 2002.
- [3] P. Bonnet, J. Gehrke, P. Seshadri, “Querying the physical world”, IEEE Personal Communications, October 2000, pp.10 – 15.
- [4] Flood Warning Technologies, <http://www.alertsystems.org> [accessed 17 December 2005].
- [5] R. Andersson, M. Sandberg, L. Urszuly, “Open Secure Office Project”, Master’s Thesis in Computer Systems and Electrical Engineering, Halmstad University, January 2005.
- [6] NAM: Network Animator, <http://www.isi.edu/nsnam/nam/> [accessed 25 May 2005].
- [7] Cygwin User’s Guide, <http://www.cygwin.com/> [accessed 25 May 2005].
- [8] M. Greis, “Tutorial for the Network Simulator NS”, <http://www.isi.edu/nsnam/ns/tutorial/index.html> [accessed 25 May 2005].
- [9] J. Chung, M. Chaypool, “NS by Example”, WPI Worcester Polytechnic Institute, Computer science, <http://nile.wpi.edu/NS/> [accessed 25 May 2005].
- [10] E. Altman, T. Jimenez, “NS Simulator for beginners”, <http://www.sop.inria.fr/maestro/personnel/Eitan.Altman/COURS-NS/n3.pdf> [accessed 25 May 2005].
- [11] The Network Simulator ns-2: Documentation, <http://www.isi.edu/nsnam/ns/ns-documentation.html> [accessed 25 May 2005].
- [12] P. Sung, A. Savvides, M.B. Srivastava, “Simulating networks of wireless sensors”, Simulation Conference, 2001, Proceedings of the Winter Volume 2, 9-12 December 2001, pp. 1330 – 1338.
- [13] J-Sim Tutorial, <http://www.j-sim.org/> [accessed 25 May 2005].
- [14] GloMoSim User Manual, Global Mobile Information Systems Simulation Library, UCLA Parallel Computing Laboratory, <http://pcl.cs.ucla.edu/projects/glomosim/> [accessed 25 May 2005].

- [15] S. Sundresh, K. Wooyoung, G. Agha, "SENS: a sensor, environment and network simulator", Simulation Symposium, 2004. Proceedings 37th Annual, 18-22 April 2004, pp. 221 – 228.
- [16] P. Levis, N. Lee, "TOSSIM: A Simulator for TinyOS Networks", Computer Science Division, University of California Berkeley, California, 17 September 2003.
- [17] P. Levis, N. Lee, M. Welse, D. Culler, "TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications", Proceedings of the ACM Conference on Embedded Networked Sensor Systems, 2003.
- [18] C. Mallanda, A. Suri, V. Kunchakarra, S.S. Iyengar, R. Kannan, A. Durrezi, "Simulating Wireless Sensor Network with OMNeT++", Sensor Network Research Group, Department of Computer Science, Louisiana State University, Baton Rouge, LA, 24 January 2005.
- [19] I. Downard, "Simulating Sensor Networks in NS-2", NRL/FR/5522--04-10073, Naval Research Laboratory, Washington, D.C., May 2004.
- [20] NRL's Sensor Network Extension to NS-2, Networks and Communication Systems Branch, U.S. Naval Research Lab,
<http://pf.itd.nrl.navy.mil/nrlsensorsim> [accessed 8 September 2005].
- [21] NS-2 Class Hierarchy,
<http://www.isi.edu/nsnam/nsdoc-classes/hierarchy.html> [accessed 25 November 2005].
- [22] The Ns-users Archives,
<http://mailman.isi.edu/pipermail/ns-users> [accessed 25 August 2005].
- [23] F. J. Ros, P. M. Ruiz, "Implementing a New Manet Unicast Routing Protocol in NS2", Department of Information and Communications Engineering, University of Murcia, Spain, December 2004,
<http://masimum.dif.um.es/nsrt-howto/html/nsrt-howto.html> [accessed 14 September 2005].
- [24] Energy Model update in ns-2,
http://www.isi.edu/ilense/software/smac/ns2_energy.html [accessed 19 October 2005].

Appendix A Basic Wireless Network Model in NS-2

The most obvious points that differ between wired and wireless models are the nodes in the network. For the wired model, the nodes are connected with links to other nodes and fixed in the specific position which is different from the wireless model when its nodes move freely and are not connected to other nodes by links. These freely moving nodes are called MobileNodes. In the NS, the C++ class MobileNode is extended from the parent class Node by adding functions of a wireless and mobile node such as ability to move within a given topology, and to receive and transmit signals to and from a wireless channel. In this chapter, we discuss the structure of MobileNode, movement/traffic scenario generation for wireless simulations, wireless network components, routing algorithms, trace support and format of wireless traces.

A1 MobileNode

MobileNode is the extension of the Node object which is created by adding functions of movement, ability to transmit and receive on a channel to communicate with other MobileNodes. The class MobileNode is derived from the C++ class Node. The mobility features such as node movement, periodic position updates, topology boundary etc are implemented in C++ while creating network components within MobileNode such as Link Layer (LL), Mac, Channel etc have been implemented in OTcl.

To configure a MobileNode, the following OTcl codes are used.

```
$ns_ node-config -adhocRouting $val(rp) \  
    -llType $val(ll) \  
    -macType $val(mac) \  
    -ifqType $val(ifq) \  
    -ifqLen $val(ifqlen) \  
    -antType $val(ant) \  
    -propType $val(prop) \  
    -phyType $val(netif) \  
    -channel $chan_1_ \  
    -topoInstance $topo \  
    -agentTrace ON \  
    -routerTrace OFF \  
    -macTrace ON \  
    -movementTrace OFF
```

This procedure creates a Mobilenode comprising; adhoc-routing protocol, network stack consisting of a link layer, mac layer, an interface queue, a network interface with an antenna using defined propagation model, channel, topography, and different tracing levels (agent, router, mac, and movement) turned on or off.

The schematic of a Mobilenode is shown below:

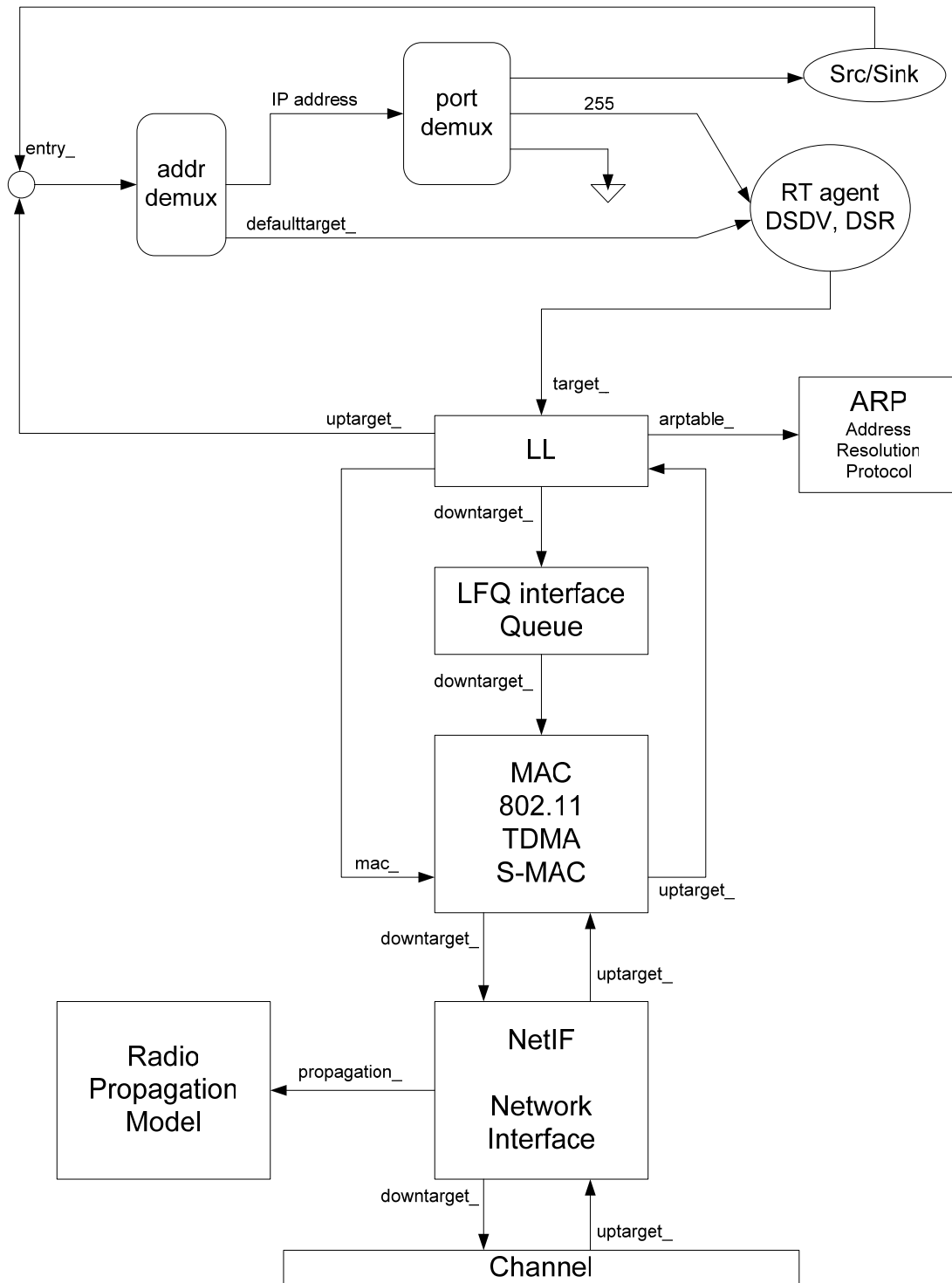


Figure A.1: The schematic of a mobile node in NS-2

A2 Node Movements

Although NS-2 supports the movement of Mobilenodes in a three dimensional topology, the Mobilenode is normally assumed to move only two dimensional in X-Y planes and Z-plane is always equal to 0. The basic mechanism to induce movement in Mobilenodes from the starting position is to give their future destinations.

To set the Mobilenode movement, the following OTcl codes are used.

```
$node set X_ <x1>
```

```
$node set Y_ <y1>
```

```
$node set Z_ <z1>
```

```
$ns at$time $node setdest <x2> <y2> <speed>
```

First, create the starting position (x1, y1), then give the sufficient time and moving speed for Mobilenode to move towards the destination (x2, y2).

The node movement generator is available in `~ns/indep-utils/cmu-scen-gen/setdest`. The node movement file can be created by using the following codes.

```
./setdest [-n num_of_nodes] [-p pausetime] [-s maxspeed] [-t simtime] [-x maxx] [-y maxy] > [outdir/movement-file]
```

The code creates a node movement scenario which consists of n nodes, and moves with maximum speed s with an average pause between the movements which is 2 seconds. The topology boundary is defined by x and y, and the simulation time is defined by t.

For the traffic scenario, random traffic of TCP and CBR connections between Mobilenodes can be generated by using a traffic scenario generator script which can be found in `~ns/indep-utils/cmu-scen-gen` and the file is `cbrgen.tcl`. The random traffic pattern can be created by using the following codes

```
ns cbrgen.tcl [-type crb|tcp] [-nn nodes] [-seed seed] [-mc connections] [-rate rate]
```

To create a random traffic pattern, we have to define the type of the traffic connection (CBR or TCP), the number of nodes and maximum number of connections, a random seed of CBR connection and a rate which is used to compute the interval time between CBR packets.

A3 Network Component in MobileNodes

To setup the network, we need to create the network stack to allow channels to access the Mobilenode. The network stack consists of a link layer (LL), ARP (Address Resolution Protocol) module connected to LL, an interface priority queue (IFq), a MAC layer, and a network interface (netIF).

The details of each component are presented below:

Link Layer: The important function of the link layer is to set the MAC destination address in the MAC header of the packet. The link layer is connected to an ARP module to resolve all IPs to hardware (MAC) address conversions. To send packets, the routing agent hands down the packets to the link layer, which then hands them down to the interface queue. To receive the packets, the MAC layer sends up the packets to the link layer, then the link layer hands them off at the node entry point. The class of LL is implemented in `~ns/ll.(cc,h)` and `~ns/tcl/lan/ns-ll.tcl`.

ARP: The ARP module receives queries from the link layer. If it knows the destination address, it will write the address on the MAC header of the packet. Otherwise, it broadcasts an ARP query, and keeps the packet temporarily. The ARP has a buffer for a single packet. If it sends an additional packet to the same destination, the earlier buffered packet will be dropped. When it knows the next hop address, the packet will be inserted into the interface queue. The class of ARPTable is implemented in `~ns-arp.(cc,h)` and `~ns/tcl/lib/ns-mobilenode.tcl`.

Interface Queue: This class gives a priority queue (to routing protocol packets), and inserts it on the head of the queue. It also supports the running of a filter queue over all packets in the queue and removes those with a specified destination address. The class of PriQueue is implemented in `~ns/priqueue.(cc,h)`.

MAC Layer: The MAC layer uses the RTS/CTS/DATA/ACK pattern for all unicast packets and sends out DATA all broadcasting packets. The class of Mac802_11 is implemented in `~ns/mac802_11.(cc,h)`

Network Interfaces: The network interface layer is used by Mobilenode to access the channel. The interface stamps each transmitted packet header with the meta-data related to the transmitting interface such as transmission power and wavelength. This meta-data is used by the propagation model in the receiving network interface to determine whether the packet has enough power to be received by the receiving node. The implementations are in `~ns/phy.(cc,h)` and `~ns/wireless-phy.(cc,h)`.

Radio Propagation Model: Friss-space attenuation ($1/r^2$) model is used for short distances and the Two Ray Ground ($1/r^4$) model is used for long distances. The implementation is in `~ns/tworayground.(cc,h)`.

Antenna: Mobilenode uses an omni-directional antenna which has unity gain. The implementation is in `~ns/antenna.(cc,h)`.

Energy Model: The energy model of a node has an initial value as “initialEnergy_”. It also has a given energy usage for every packet it transmits and receives as “txPower_” and “rxPower_”. The initial energy will be decreased by energy usage for every transmitted and received packet. When the energy level at the node decreases to zero, the node cannot receive or transmit the packet. The energy model is implemented in `~ns/energymodel.(cc,h)`. The new energy model is extended in [24]. This model is turned into the sleep mode to save energy.

A4 Different Types of Routing Agents in Mobile Networking

There are currently four ad-hoc routing protocols, DSDV, DSR, AODV, and TORA, implemented for mobile networks in NS. The implementations which are mainly in C++ can be found in `~ns/tcl/mobility/dsdv.tcl`, `dsr.tcl`, `tora.tcl`, `~ns/tcl/lib/ns-lib.tcl` and `~ns/dsr`, `tora`, and `aodv`.

A5 Trace support

There are three types of trace object, CMUTrace/Drop, CMUTrace/Recv and CMUTrace/Send, which are used for tracing dropped, received and sent packets by agents, routers, MAC layers or interface queues in NS. The implementation methods and procedures are in `~ns/trace(cc,h)` and `~ns/tcl/lib/ns-cmutrace.tcl`.

A6 Wireless Trace Formats

The trace file is an important output of the simulation to analyze the details of networks. The wireless simulations use two different trace formats, which are described in this part by using the output traces.

A6.1 Normal trace format

An example of a line in output trace is

```
r 5.009402153 _2_ AGT --- 0 cbr 230 [13a 2 1 800] ----- [1:0 2:0 30 2] [0]
```

- The first field is a letter which can be one of the types r, s, f and D for “received”, “sent”, forwarded” and “dropped” respectively. It can be M when a node has a movement a location.
- The second field is the time in second.
- The third field is the node number.
- The fourth field is the type that the packet that can be one of MAC, AGT, RTR and IFQ. MAC indicates a MAC layer, AGT indicates a transport layer packet, RTR indicates a routed packet and IFQ stands for the event which relates to the interference priority queue.
- The number represents after the dashes is the global sequence number of the packet.
- The next field concerns the information of the packet type (tcp, ack or udp).
- The next number is the packet size in bytes.
- The four numbers in the first bracket stand for the MAC layer information. The first hexadecimal number specifies the expected time in seconds to send this data packet over the wireless channel. The second number is the MAC-id of the sending node. The third number is the MAC-id

of the receiving node. The last number stands for the MAC type, while the number “800” refers to ETHERTYPE_IP.

- The next four numbers in the second bracket concern the IP source and destination addresses, and the TTL (Time to Live) of the packet.

- The third bracket concerns the tcp information; its sequence number and the acknowledgement number.

A6.2 Revised format for wireless traces

The revised format is improved to merge a wireless trace. This new trace support is now only available for wireless simulations and will be extended to the rest of NS-2 in the future. The following command is used to create this new format.

```
$ns use-newtrace
```

An example of this new trace format is shown below:

```
S -t 0.267662078 -Hs 0 -Hd -1 -Ni 0 -Nx 5.00 -Ny 2.00 -Nz 0.00 -Ne  
-1.000000 -N1 RTR -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 0.255 -Id -1.255 -It  
message -Il 32 -If 0 -Ii 0 -Iv 32
```

```
r -t 100.004776054 -Hs 0 -Hd -1 -Ni 1 -Nx 25.05 -Ny 20.05 -Nz 0.00 -Ne  
-1.000000 -N1 AGT -Nw --- -Ma a2 -Md 1 -Ms 0 -Mt 800 -Is 0.0 -Id -1.0 -It  
tcp -Il 1020 -If 2 -Ii 21 -Iv 32 -Pn tcp -Ps 0 -Pa 0 -Pf 1 -Po 0
```

The features of the new trace format are described below:

Event type: The first field describes the event type taking place at the node which can be one of the four types: s (send), r (receive), d (drop) and f (forward)

General tag: The second field starting with “-t” may stand for time or global setting.

Node property tags: This field describes the node properties like node-id, trace level such as router or MAC. The tags that start with a leading “-N” are:

- | | |
|-----------------------------|--|
| - Ni: node id | - Nx: node’s x-coordinate |
| - Ny: node’s y-coordinate | - Nz: node’s z-coordinate |
| - Ne: node energy level | - NI: trace level, such as AGT, RTR, MAC |
| - Nw: reason for the event. | |

Packet information at IP level: This field starts with “-I” and the details are:

- | | |
|---|-------------------------------------|
| - Is: source address.source port number | - Id: dest address.dest port number |
| - It: packet type | - Il: packet size |
| - If: flow id | - Ii: unique id |
| - Iv: ttl value | |

Next hop info: This field, which describes next hop info, starts with “-H”

- Hs: id for this node
- Hd: id for next hop towards the destination

Packet info at MAC level: This field, which describes MAC layer information, starts with “-M”

- Ma: duration
- Md: dst’s Ethernet address
- MS: src’s Ethernet address
- Mt: Ethernet type

Packet info at Application level: This field describes the packet information at the application level which consists of the type of application like ARP, TCP, and different types of routing protocol like DSDV, DSR, AODV etc. This field start with “-P”

- P arp: For ARP, whose details are:
 - Po: ARP Request/Reply
 - Pm: src mac address
 - Ps: src address
 - Pa: dst mac address
 - Pd: dst address
- P dsr: The DSR routing protocol details are:
 - Pn: how many nodes traversed
 - Pq: routing request flag
 - Pi: route request sequence number
 - Pp: routing reply flag
 - Pl: reply length
 - Pe: src of srcrouting, dst of the source routing
 - Pw: error report flag
 - Pm: number of errors
 - Pc: report to whom
 - Pb: link error from link a to link b
- P cbr: The details of the CBR application are:
 - Pi: sequence number
 - Pf: how many times this pkt was forwarded
 - Po: optimal number of forwards
- P tcp: The details of TCP flow are:
 - Ps: seq number
 - Pa: ack number
 - Pf: how many times this pkt was forwarded
 - optimal number of forwards

This field is under development, which new tags will be added in the future.

See [8], [10] and [11] for more information.

Appendix B Implementation Codes for Wireless Sensor Network Classes

B1 Environment Module Command()

```

ENV::command(int argc, const char*const* argv) {
    if(argc == 2) {
        Tcl& tcl = Tcl::instance();

        if(strncasecmp(argv[1], "id", 2) == 0) {
            tcl.resultf("%d", index);
            return TCL_OK;
        }

        if(strncasecmp(argv[1], "start", 2) == 0) {
            htimer.handle((Event*) 0);
            return TCL_OK;
        }
    }
    else if(argc == 3) {

        if(strcmp(argv[1], "pulserate") == 0) {
            Hello_Interval = atof(argv[2]);
            return TCL_OK;
        }
        else if(strcmp(argv[1], "env") == 0)
        {
            // Envenon types are #defined as ints in env_packet.h
            if (!strcmp("TEST_ENV", argv[2]))
            {
                ENV::Env = TEST_ENV;
            }
            else if (!strcmp("CO", argv[2]))
            {
                ENV::Env = CO;
            }
            else if (!strcmp("HEAVY_GEO", argv[2]))
            {
                ENV::Env = HEAVY_GEO;
            }
            else if (!strcmp("LIGHT_GEO", argv[2]))
            {
                ENV::Env = LIGHT_GEO;
            }
            else if (!strcmp("SOUND", argv[2]))
            {
                ENV::Env = SOUND;
            }
        }
    }
}

```

```
    }
    else if (!strcmp("OS", argv[2]))
    {
        ENV::Env =OS
    }
else
    {
        fprintf(stderr, "Unknown Env sensed: %s. Continueing...\n",
            argv[2]);
    }
    return TCL_OK;
}
else if(strcmp(argv[1], "target") == 0) {
    Tcl& tcl = Tcl::instance();
    tcl.evalf("%s set node_", this->name());
    tcl.evalf("%s set ifq_(0)", tcl.result());
    ifqueue = (PriQueue*) TclObject::lookup(tcl.result());
    return Agent::command(argc, argv);
}
else {
    return TCL_OK;
}
}
return Agent::command(argc, argv);
}
```

```
// Constructor
ENV::ENV(nsaddr_t id) : Agent(PT_ENV), htimer(this),
    Hello_Interval(1), Env(TEST_ENV) {
    srand(10);
    envDebugValue=0;
    SetDebugLevel(0);
    OpenDebugLog("tempenv");
    index = id;
    seqno = 0;
    logtarget = 0;
    smoothQueueLength = 0;
}
```

```
void
ENVHelloTimer::handle(Event*) {
    if(doSchedule) {
        Scheduler::instance().schedule(this, &intr, 0);
        doSchedule=0;
    }
    else{
```

```

    jitter = 0.01 * Random::uniform();      // jitter helps avoid collisions
    doSchedule=1;
    agent->emanate();
    Scheduler::instance().schedule(this, &intr, agent->Hello_Interval + jitter);
}
}

```

B2 Environment Module Emanate()

```

ENV::emanate() {
    seqno++; // need to add wrap around functionality to both the seqno sending and teh
    recieving side
    Packet *p = Packet::alloc();
    struct hdr_cmh *ch = HDR_CMH(p);
    struct hdr_ip *ih = HDR_IP(p);
    struct hdr_env_main *mh = HDR_ENV(p);
    struct payload *msg_payload = ENV_PAYLOAD(p, sizeof(struct hdr_env_main)); //in bytes

    DMSG(8, "%d seqno being packed \n", seqno);
    fflush(stdout);

    mh->pck_length=msg_payload->size()+sizeof(struct hdr_env_main); // in bytes

    msg_payload->msg_oAddr=here_.addr_;
    msg_payload->msg_size=sizeof(struct payload); // will point over be careful
    msg_payload->msg_seqno=seqno;
    msg_payload->Env = ENV::Env;

    ch->ptype() = PT_ENV;

    ih->saddr() = here_.addr_;
    ih->daddr() = IP_BROADCAST;
    ih->sport() = ENV_PORT;
    ih->dport() = ENV_PORT;
    ih->tttl_ = 1;

    DMSG(8, "sending Hello from %d at %.4f\n", here_.addr_, CURRENT_TIME);
    fflush(stdout);
    Scheduler::instance().schedule(target_, p, 0.0);
    // target_->EnvAgent::recv(p);
}

```

B3 ns-mobilenode.tcl

```
// set sensing energy for sensor nodes
Node/MobileNode instproc setPs { val } {
    $self instvar netif_
    if [info exists netif_(1)] {
        $netif_(1) setSensePower $val
    }
}

//setup up link layer, mac layer, network interface and physical layer structures for a
sensor node
Node/MobileNode instproc add-ENVinterface { channel pmodel lltype mactype \
    qtype qlen iftype anttype inerrproc outerrproc fecproc} {
    $self instvar nifs_ netif_ mac_ ifq_ ll_

    # puts "add-ENVinterface: SENSORchannel_ is $channel\n"
    set ns [Simulator instance]
    set imepflag [$ns imep-support]
    set t $nifs_
    incr nifs_

    # this hack prevents MacIndex count from incrementing twice for
    # a (multi-homed) sensor node (see mac.cc)
    if { $t > 0 } {
        $mac_([expr $t-1]) adjust_index
    }

    set netif_($t) [new $iftype] ;# interface
    set mac_($t) [new $mactype] ;# mac layer
    set ifq_($t) [new $qtype] ;# interface queue
    set ll_($t) [new $lltype] ;# link layer
    set ant_($t) [new $anttype] ;# antenna type
    set namfp [$ns get-nam-traceall]
    #
    # Local Variables
    #
    set nullAgent_ [$ns set nullAgent_]
    set netif $netif_($t)
    set mac $mac_($t)
    set ifq $ifq_($t)
    set ll $ll_($t)

    #
    # Link Layer
    #
    $ll mac $mac
```

```

$ll down-target $ifq

#
# Interface Queue
#
$ifq target $mac
$ifq set limit_ $qlen
set drpT [cmu-trace Drop "IFQ" $self]
$ifq drop-target $drpT

if { $namfp != "" } {
    $drpT namattach $namfp
}
#
# Mac Layer
#
$mac netif $netif
$mac up-target $ll
$netif channel $channel
$netif up-target $mac
$netif propagation $pmodel           ;# Propagation Model
$netif node $self                     ;# Bind node <---> interface
$netif antenna $ant_($t)
#
# Physical Channel
#
$channel addif $netif

if { [Simulator set MacTrace_] == "ON" } {
    #
    # Trace Received Packets
    #
    if {$imepflag != ""} {
        set rcvT [$self mobility-trace Recv "MAC"]
    } else {
        set rcvT [cmu-trace Recv "MAC" $self]
    }
    $rcvT target [$mac up-target]
    $mac up-target $rcvT
    if { $namfp != "" } {
        $rcvT namattach $namfp
    }
    #
    # Trace Dropped Packets
    #
    if {$imepflag != ""} {
        set drpT [$self mobility-trace Drop "MAC"]
    }

```

```
        } else {
            set drpT [cmu-trace Drop "MAC" $self]`
        }
        $mac drop-target $drpT
        if { $namfp != "" } {
            $drpT namattach $namfp
        }
    } else {
        $mac log-target [$ns set nullAgent_]
        $mac drop-target [$ns set nullAgent_]
    }
}

# set sensing energy for sensor nodes
Node/MobileNode instproc setPs { val } {
    $self instvar netif_
    if [info exists netif_(1)] {
        $netif_(1) setSensePower $val
    }
}
```

B4 cmu-trace.h,cc

```
void
CMUTrace::format_env(Packet *p, int offset)
{
    struct payload *msg_payload;
    struct hdr_env_main *mh = HDR_ENV(p);
    struct hdr_cmh *ch = HDR_CMH(p);
    struct hdr_ip *ih = HDR_IP(p);
    struct hdr_env_hello *asymlink, *symlink, *mprlink, *hdr_hello;
    struct hdr_env_tc *hdr_tc;
    int local_offset=sizeof(struct hdr_env_main);

    msg_payload=ENV_PAYLOAD(p, local_offset);

    switch(msg_payload->environment)
    {
        case TEST_ENV:
            sprintf(pt_->buffer() + offset, "[%d %d %d %d] (TEST_ENV)",
                msg_payload->msg_oAddr, msg_payload->msg_seqno,
                msg_payload->msg_size, msg_payload->phenomenon);
            break;
        case C0:
            sprintf(pt_->buffer() + offset, "[%d %d %d %d] (C0)",
                msg_payload->msg_oAddr, msg_payload->msg_seqno,
```

```
        msg_payload->msg_size, msg_payload->environment);
    break;
case HEAVY_GEO:
    sprintf(pt_->buffer() + offset, "[%d %d %d %d] (HEAVY_GEO)",
        msg_payload->msg_oAddr, msg_payload->msg_seqno,
        msg_payload->msg_size, msg_payload->environment);
    break;
case LIGHT_GEO:
    sprintf(pt_->buffer() + offset, "[%d %d %d %d] (LIGHT_GEO)",
        msg_payload->msg_oAddr, msg_payload->msg_seqno,
        msg_payload->msg_size, msg_payload->environment);
    break;
case SOUND:
    sprintf(pt_->buffer() + offset, "[%d %d %d %d] (SOUND)",
        msg_payload->msg_oAddr, msg_payload->msg_seqno,
        msg_payload->msg_size, msg_payload->environment);
    break;

    default:
#ifdef WIN32
        fprintf(stderr, "CMUTrace::format_adv: invalidENV packet type\n");
#else
        fprintf(stderr, "%s: invalid Environment packet type \n", __FUNCTION__);
#endif
    abort();
}
}
```


Appendix C Example Tcl Codes for Testing Routing Protocols

The example Tcl codes for testing AODV routing protocol with 49 sensor nodes in Chapter 3.4 are shown below:

```
# =====
# Define options
# =====
set val(chan)          Channel/WirelessChannel    ;# channel type
set val(prop)          Propagation/TwoRayGround   ;# radio-propagation model
set val(netif)         Phy/WirelessPhy           ;# network interface type
set val(mac)           Mac/802_11                ;# MAC type
set val(ifq)           Queue/DropTail/PriQueue    ;# interface queue type
set val(ll)            LL                        ;# link layer type
set val(ant)           Antenna/OmniAntenna       ;# antenna model
set val(ifqlen)        50                       ;# max packet in ifq
set val(nn)            59                       ;# number of mobilenodes
set val(rp)            AODV                     ;# routing protocol
set val(x)             1100                     ;# grid width
set val(y)             1100                     ;# grid hieght

Queue/DropTail/PriQueue set Prefer_Routing_Protocols    1

# specify the transmit power
Phy/WirelessPhy set Pt_ 0.1

puts "This is an AODV routing protocol testing in a sensor network"

# =====
# Main Program
# =====

#
# Initialize Global Variables
#

set ns_ [new Simulator]
set tracefd [open 49AODV.tr w]
$ns_ trace-all $tracefd

set namtrace [open 49AODV.nam w]
$ns_ namtrace-all-wireless $namtrace $val(x) $val(y)

# set up topography object
set topo [new Topography]
```

```
$topo load_flatgrid $val(x) $val(y)

#
# Create God
#
set god_ [create-god $val(nn)]
$god_ off
$god_ allow_to_stop
$god_ num_data_types 1

#configure sensor channel and wireless channel
set chan_1_ [new $val(chan)]
set chan_2_ [new $val(chan)]

# configure environment node
set val(rp) ENV                                ;# ENV routing protocol
$ns_ node-config \
    -adhocRouting $val(rp) \
    -llType $val(ll) \
    -macType $val(mac) \
    -ifqType $val(ifq) \
    -ifqLen $val(ifqlen) \
    -antType $val(ant) \
    -propType $val(prop) \
    -phyType $val(netif) \
    -channel $chan_1_ \
    -topoInstance $topo \
    -agentTrace OFF \
    -routerTrace OFF \
    -macTrace ON \
    -movementTrace OFF

    for {set i 0} {$i < 10} {incr i} {          ;#set environment node
        set node_($i) [$ns_ node]                ;#from node 0 to node 9
        $node_($i) random-motion 0                ;#total 10 nodes
        $god_ new_node $node_($i)
        $node_($i) namattach $namtrace
        $ns_ initial_node_pos $node_($i) 25
        [$node_($i) set ragent_] pulserate 2.0    ;#transmit pulse every 2 sec
    }
```

```
# configure sensor nodes
set val(rp) AODV                                ;# AODV routing protocol
$ns_ node-config \
    -adhocRouting $val(rp) \
    -channel $chan_2_ \
    -SENSORchannel $chan_1_

    for {set i 10} {$i < 59 } {incr i} {        ;#set sensor node
        set node_($i) [$ns_ node]              ;#from node 10 to node 58
        $node_($i) random-motion 1              ;#total 49 nodes
        $god_ new_node $node_($i)
        $node_($i) namattach $namtrace
    }

# configure user node
set val(rp) AODV                                ;# AODV routing protocol
$ns_ node-config \
    -adhocRouting $val(rp) \
    -channel $chan_2_ \
    -SENSORchannel "off"

    for {set i 59} {$i < 60 } {incr i} {        ;#set node 59 ->user node
        set node_($i) [$ns_ node]
        $node_($i) random-motion 1
        $god_ new_node $node_($i)
        $node_($i) namattach $namtrace
    }

#
# Provide initial (X,Y, for now Z=0) co-ordinates for mobilenodes
#

#import the "randommovement" file which create and set 10 environment nodes
#to have random movement
source.orig "randommovement"

#import the "49nodes" file which create 49 sensor and one user nodes
source.orig "49nodes"
```

```
#####
# Attach the sensor agent to the sensor node, and build a conduit thru which
# recieved Env packets will reach the sensor agent's recv routine

# attach a Sensor Agent (i.e. sensor agent) to sensor nodes
for {set i 10} {$i < 59} {incr i} {
    set sensor_($i) [new Agent/SensorAgent]
    $ns_ attach-agent $node_($i) $sensor_($i)
}

# specify the sensor agent as the up-target for the sensor node's link layer
# configured on the sensor interface
for {set i 10} {$i < 59} {incr i} {
    [$node_($i) set ll_(1)] up-target $sensor_($i)
    $ns_ at 0.01 "$sensor_($i) start"
}

#####

# setup UDP connections to a user node, and attach sensor apps
set sink_0 [new Agent/UDP]
$ns_ attach-agent $node_(59) $sink_0
for {set i 10} {$i < 59} {incr i} {
    set src0_($i) [new Agent/UDP]
    $ns_ attach-agent $node_($i) $src0_($i)
    $ns_ connect $src0_($i) $sink_0

    set app0_($i) [new Application/SensorApp]
    $app0_($i) attach-agent $src0_($i)
}
for {set i 10} {$i < 59} {incr i} {
    $ns_ at 1.0 "$app0_($i) start $sensor_($i)"
}

#Tell nodes when the simulation ends
#
for {set i 0} {$i < 59} {incr i} {
    $ns_ at 20.0 "$node_($i) reset";
}

$ns_ at 20.1 "stop"
$ns_ at 20.1 "puts \"NS EXITING...\" ; $ns_ halt"
```

```
proc stop {} {  
    global ns_ tracefd namtrace  
    $ns_ flush-trace  
    close $tracefd  
    close $namtrace  
}
```

```
#Begin command line parsing
```

```
puts "Starting Simulation..."  
$ns_ run
```

For Testing DSDV routing protocol, the above codes are changed only in the parameter of `val(rp)` from AODV to DSDV.